How to Improve the Security Skills of Mobile App Developers

An Analysis of Expert Knowledge

Charles Weir

Security Lancaster

School of Computing and Communications

Lancaster University, UK

A thesis submitted for the degree of

Masters by Research

February 2017

With thanks to my wife Julia

The best thing for being sad... is to learn something. That is the only thing that never fails. You may grow old and trembling in your anatomies, you may lie awake at night listening to the disorder of your veins, you may miss your only love, you may see the world about you devastated by evil lunatics, or know your honour trampled in the sewers of baser minds. There is only one thing for it then – to learn. Learn why the world wags and what wags it. That is the only thing which the mind can never exhaust, never alienate, never be tortured by, never fear or distrust, and never dream of regretting. Learning is the thing for you. Look at what a lot of things there are to learn—pure science, the only purity there is. You can learn astronomy in a lifetime, natural history in three, literature in six. And then, after you have exhausted a million lifetimes in biology and medicine and theocriticism and geography and history and economics – why, you can start to make a cartwheel out of the appropriate wood, or spend fifty years learning to begin to learn to beat your adversary at fencing. After that you can start again on mathematics, until it is time to learn to plough.

White, T. H. (1939). The Once and Future King.

Declaration

This thesis is entirely my own work, and has not been submitted in any form for the award of a higher degree elsewhere. Ethics approval was granted for this research according to university guidelines. Some of the ideas in this thesis were the product of discussion with my supervisors Professor Awais Rashid and Professor James Noble.

This thesis includes materials from the following peer-reviewed published work written by myself:

Early Report: How to Improve Programmers' Expertise at App Security?

Weir, C., Rashid, A. & Noble, J. 8/04/2016 *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security: Co-located with ESSoS 2016*. Aspinall, D., Cavallaro, L., Seghir, M. & Volkamer, M. (eds.). London, UK: CEUR-WS.org, Vol. 1575, p. 49-50

How to Improve the Security Skills of Mobile App Developers: Comparing and Contrasting Expert Views

Weir, C., Rashid, A. & Noble, J. 7/06/2016 *Proceedings of the 2016 ACM Workshop on Security Information Workers*. Biddle, R., Chu, B., Lipford, H. & Murphy-Hill, E. (eds.). New York: ACM

Reaching the Masses: A New Subdiscipline of App Programmer Education

Weir, C., Rashid, A. & Noble, J. *Visions and Reflections, Foundations of Software Engineering Conference 2016.* 13/11/2016 ACM

Charles Weir

Summary

Much of the world relies heavily on apps. Increasingly those apps handle sensitive information: controlling our financial transactions, enabling our personal communication and holding intimate details of our lives. So the security of those apps is becoming increasingly vital. Yet research shows that those apps contain frequent security and privacy problems; and that almost all of these issues could have been avoided had the developers had sufficient motivation, support and knowledge. This lack of developer knowledge and support is widely perceived as a major threat.

We therefore investigated the skills, approach and motivation required for developers. We conducted a Constructivist Grounded Theory study, involving face-to-face interviews with a dozen experts whose cumulative experience totalled over 100 years of secure app development, to develop theory on secure development techniques. The study identified that the subdiscipline of app development security is still at an early stage, and found surprising discrepancies between current industry understanding and the experts' recommendations. In particular it found that a secure development process tends not to appeal to app developers; and that the approach of identifying common types of security problems is too limited to give an effective security solution.

Instead we identified a set of successful techniques we call 'Dialectical Security', where 'dialectic' means learning by questioning. These techniques use dialogue with a range of counterparties to achieve app security in an effective and economical way. The security increase comes from continued dialog, not passive learning.

The novel contribution of our work is to provide:

- A grounded theory of secure app development that challenges conventional processes and checklists, and
- A shift in perspective from process to dialectic.

Only by working to develop the Dialectical Security skills of app developers shall we begin to see the kinds of secure apps we need to combat crime and privacy invasions.

Acknowledgements

Thank you first to my supervisors, Awais Rashid and James Noble, especially for their support for writing conference papers, which have led not only to interesting ideas in their own right but also to suggestions and to ideas from other academics.

Thank you to all those who generously gave time to be interviewed; their wisdom and insight were essential to this work.

Thank you to my wife, Julia, for her support both academic and practical, and for originally suggesting this one-year research project.

Thank you to Kayla Friedman and Malcolm Morgan for the Microsoft Word thesis template used as a basis for the document, and to Gerald Kotonya and Debbie Stubbs for help changing it and distributing it for Lancaster use.

Thank you to my fellow post-graduate students at Security Lancaster for much help and support, and especially to Syed Asad Ali Naqvi for his help with the intricacies of Grounded Theory.

Contents

List of Tables

List of Figures

List of Abbreviations and Acronyms

ACM	Association for Computing Machinery
API	Application Programming Interface
BCS	British Computer Society
CA	Certification Authority
CCS	Conference on Computer and Communications Security
CHI	Computer Human Interaction
CISO	Chief Information Security Officer
CMM	Capability Maturity Model
CSCW	Conference on Computer-Supported Cooperative Work
DARPA	US Defense Advanced Research Projects Agency
EMV	Eurocard, MasterCard and Visa
ENISA	European Union Agency for Network and Information Security
ESEC	European Software Engineering Conference
ESSoS	International Symposium on Engineering Secure Software and Systems
FIPS	Federal Information Processing Standards
FSE	Foundations of Software Engineering Conference
GT	Grounded Theory
HCC	Human-Centric Computing
HCI	Human Computer Interaction
HIDS	Host-based Intrusion Detection System
HTTPS	Hypertext Transfer Protocol over SSL
ICSE	International Conference on Software Engineering
ID	Identifier
IDE	Interactive Development Environment
IEC	International Electrotechnical Commission

IEEE	Institute of Electrical and Electronics Engineers
iOS	iPhone OS
IP	Intellectual Property
ISO	International Standardization Organization
ISP	Internet Service Provider
IT	Information Technology
JEE	Java Enterprise Edition
LLVM	Low Level Virtual Machine
MITM	Man In The Middle
MOOC	Massively Open Online Course
MSDN	Microsoft Developer Network
NFC	Near Field Communication
OID	Object Identifier
OOD	Object Oriented Design
OS	Operating System
OSSEC	Open Source HIDS SECurity (an open-source tool)
OWASP	Open Web Application Security Project
PDF	Portable Document Format
PIN	Personal Identification Number
SANS	SysAdmin, Audit, Network, and Security (an institute)
SIGSAC	Special Interest Group on Security, Audit & Control
SIGSOFT	Special Interest Group on Software
SPI	Software Process Improvement
SQL	Structured Query Language (for databases)
SSL	Secure Sockets Layer
TED	Technology, Entertainment and Design (non-profit organisation)

UML	Unified Modelling Language (for software design)
WOOT	USENIX Workshop On Offensive Technologies
XML	Extensible Mark-up Language

1 Introduction

The past ten years has seen a massive growth in the creation and usage of mobile phone and tablet apps. We use apps to communicate, apps to plan, apps to manage our finances, apps to do our shopping, and apps to remember all our security information. Increasingly those apps are handling sensitive information about us: controlling our financial transactions, enabling our personal communication and social networking and holding the intimate details of our lives. So the security of those apps is becoming increasingly vital.

Creating our apps are more than 2.9 million app developers, of whom only some 25% are professionals developing apps for companies [106]. In these apps, cloud-based connectivity and social networking functionality are making security and privacy issues fundamentally important. So security expertise – and hence effective security practices – in those developing such apps is vital.

Yet there is considerable evidence that such expertise is lacking. Analysis of the top five payment apps by Bluebox [18], a security solution provider, found significant security failures in each; analysis of a range of Android apps by Enck et al. [39] found privacy problems in most of them. Both analyses highlighted that it was the choices that the app programmers had made that were causing the problems; given the same environment and cloud services they could have chosen problem-free implementations.

Indeed a recent IBM-driven survey of opinions about app security in American companies [84] revealed that more than 70% percent believed that the developer inexperience was a major threat to their business.

Thus the security of users and data depends vitally on programmers' security practices. Therefore, it is important to improve the effectiveness of developers at producing secure apps. We can choose from three possible research questions to address this, all worthwhile:

(1) What kinds of security errors do programmers make?
(2) How do we improve the systems and compilers that support the developers in their work; and
(3) How can we improve the security skills of the app developers themselves?

There has been a good deal of work on the first question such as the previously-mentioned work by Enck and Bluebox, or work by Xie et al. [112] exploring the reasons why programmers make security errors. Various projects – including Xie et al.'s IDE enhancements [111], compiler improvements and libFuzzer's testing support [66] – address the second question. Taking the third question, however, there is little understanding how and why app programmers learn security and what approaches are likely to work best.

This work examines that third question. In the remainder of this chapter we shall explore briefly the state-of-the-art, derive research questions and briefly discuss the approach used, highlight novel aspects of the work and its results, and finally outline the remainder of the thesis.

1.1 State of the Art and Its Limitations

The study of app programmer education and empowerment involves research on

1. how programmers actually learn,
2. how they can be helped to learn, and
3. what they should learn.

Taking each of these aspects of research, there is a small amount of relevant work on how app developers learn security. Balebako et al. [10] surveyed and interviewed over 200 app developers, and concluded most approached security issues using web search, or by consulting peers. A survey by Acar et al. concluded the same; and they also determined experimentally the surprising result that programmers using digital books achieved better security than those using web search [2]. Yskout et al. [116] tested experimentally the effect of using security patterns in server design; the results suggested a benefit but were statistically inconclusive.

Considering how programmers can be helped to learn, two projects, Xie et al. [111] and Nguyen et al. [77], have both developed prototype IDE-based tools to teach programmers by detecting possible security flaws in Android developments. One might expect, however, that the most effective approach would be a prescriptive set of instructions to programmers what to do, a 'Secure Development Lifecycle' such as those promoted by Microsoft [69] and others. However Conradi and Dyba [26] identified that programmers have difficulty with, and resist learning from, formal written routines. This conforms to the author's own experience; he has never encountered an app development team voluntarily adopting a process of that kind. That suggests that app developers need to find a more lightweight, less prescriptive approach.

Considering what programmers need to learn, there is a good deal of general knowledge available on software security, such as books by Anderson, by Pfleeger and by Schneier [7,82,93] though these work at a level that is rarely helpful for app developers. Knowledge useful to developers derives from a variety of sources. There is the security patterns movement [71], which showed promise but has not been widely adopted [115]; there are books written by software security practitioners (e.g. [55,68]), which tend to be helpful but are rarely adopted by app programmers – possibly for the reasons identified by Conradi and Dyba above. Then there are 'black hat' books, such as the 'Android Hacker's Handbook' [35], describing possible attacks and sometimes suitable mitigations for programmers to put against them. These seem to sell well, but work only at a technical level; they do not address the kind of security or privacy issues that derive from the app domain rather than the technicalities of app programming.

Lastly there are web question-and-answer sites and operating system specific websites and books on app security, which are consulted by developers but suffer from the same problem – as well as, in the case of the question-and-answer sites, doubtful accuracy [2].

Consistent in all this literature, whether the patterns books, the security practitioners, the black hat literature or websites, is an emphasis on artefacts; for them processes are ways to deliver those artefacts. Thus all talk in terms of documents (assessments, architectures, plans) and aspects of system design. Each source leaves the development team to make their own decisions how to achieve those artefacts; there has been little research into how the teams might do this. Moreover whilst 'whole system security' experts such as Anderson [7] are excellent at driving a holistic, rather than purely technical, view of software security, they rarely consider the team interactions needed to achieve the results they need.

Chapter 3 explores the literature on programmer learning in more detail; chapter 4 discusses learning resources available to programmers.

1.2 Thesis Objectives

Our research model assumes a self-motivated development team of one or more developers who are empowered to make their own decisions on development process, tools and philosophy. In our experience this model reflects common practice in all but the most disciplined corporate and organisational cultures. The impact of making their own decisions is important; it means that techniques which are successful but unattractive or demotivating for developers are unlikely to be of value, since they will not be adopted.

We are also interested only in individuals and teams working on software that does have security and privacy implications. Clearly a team creating a stand-alone game with no external communication will not need to worry about security; so we do not need to convert the entire developer world into security fanatics. However as the research above highlights, many more apps have security – and especially privacy – implications than developers may be aware of, and there is a need to consider how to encourage developers to address these.

The research question of this thesis is therefore:

RQ1 What techniques and ideas will appeal to development teams and lead to them developing more secure app software?

Our purpose in the research was to generate knowledge about good ways to develop apps securely. We started the research with no preconceptions as to whether we were looking for a single solution or diverse suite of solutions. Our research approach was driven by two perceptions:

- We had found few resources indicating how to tackle app development security.
- Existing literature tended to be negative in approach, listing things the developer must not do; this contrasts with the kinds of books preferred by developers which tend to be positive in outlook.

Since we had no initial theory to test, we considered an experimental approach to be unsuitable. For the same reason we ruled out surveys to test hypotheses. Instead we wanted to generate theory, based on an exploration of a range expert knowledge of existing practice. We therefore chose a Grounded Theory (GT) approach [21,47], since GT has been used extensively for that purpose in Software Engineering research [100].

Our major resource was personal connections and links to industry specialists in app development, including in secure app development. Thus our Grounded Theory study used semi-structured interviews over 6 months with a dozen such experts, whose cumulative experience totalled more than 100 years of secure app development. To encourage positivity, we used elements of Appreciative Inquiry [27] in our questioning: the 'Discovery' of best practice and the 'Dream' of ideal practice.

The original research question leads to a number of further questions:

RQ2 What motivated the experts themselves to learn software security; how did they do so; and how do they continue to learn?

RQ3 What are the most effective techniques to deliver app security?

RQ4 How should we effectively introduce security to app development teams?

These further questions motivate the different questions used in the survey, as discussed in section 2.12.

1.3 Novel Contributions

From our research we show in chapter 5 that there is little similarity in people's motivation to learn software security, nor consensus on how to motivate app developers to do so. We also find diverging opinions on the use of teamwork and on the best approach to implementing security. From these findings, by comparison with the history of other paradigms, we suggest in section 5.7 that the subdiscipline of app development security is at an early stage of development.

Furthermore, we show in chapter 6 that developers need to have a wider view of app security than merely code-level technical issues, and in section 6.1 that the experts' recommendation is to use 'dialectic' techniques, seeking out and responding to challenges from a variety of counterparties. We identify six such techniques of 'Dialectical Security', and explore each in some detail in chapter 7.

Finally, in chapter 8, we offer a range of original 'interventions' capable of teaching such techniques even to such a disparate group as solo app developers, and offer possible research programs to investigate both the techniques of Dialectical Security and the interventions in more detail.

1.4 Conventions in this Thesis

This thesis covers aspects of best practice for app development teams on both security and privacy. To save cumbersomeness in the text, we have referred throughout to 'app security' to cover both these aspects; where there is a distinction, we make it plain which to which we are referring. Similarly a decision in the context of the wider goal for the software might be a 'commercial decision' for a profitmaking company, or a 'political decision' for a development team in a government department; for convenience this thesis refers to both as a 'commercial decision'.

This thesis follows the convention of many academic papers in using 'we' to refer to the author working in conjunction with both supervisors; the thesis refers to Charles Weir the individual in the third person, as 'the author'.

1.5 Thesis Overview

The following chapters in this work describe the results of the Grounded Theory study. They explore existing work on the subject; analyse surveys with a dozen experts of different types in the app security field; highlight the differences between approaches from different experts; discuss the range of different resources; synthesises six important techniques for developers to learn; and conclude with suggestions of further work considering how they might best be taught. Specifically, the chapters are as follows:

Chapter 2 introduces the Grounded Theory research method used, explaining the philosophy used, how the interviewees were chosen, what type of people they were and how the results were analysed.

Chapter 3 explores existing research literature addressing programmer education and app security.

Chapter 4 extends the literature review to cover resources available to programmers to learn about app software security.

Chapter 5 analyses our findings from our interviews, exploring the differences amongst interviewees on motivation and approach.

Chapter 6 explores the best techniques for app security, highlighting the ‘Dialectical’ nature of good app security; and introduces three personas representing different kinds of app developer.

Chapter 7 discussing six specific techniques of Dialectical Security, illustrated with specific examples using the personas, and including detail from the interviews.

Chapter 8 explores possible ways to extend this work, with a vision on approaches to improve the security behaviour of app programmers generally.

2 Research Process

2.1 Introduction

This chapter explores the methods used in the research process. It introduces the author and aims, and outlines the two foundations to the research: Grounded Theory and Appreciative Inquiry. It also describes the interview process, the interviewees, and the two forms of information derived from them.

2.2 Research Philosophy

A good starting point is to define the philosophical approach we have as researchers. Creswell [30] describes four major philosophical stances depending on the aims and needs of the researcher. Briefly these are *positivist*, looking for a single objective truth; *relativist*, rejecting the idea that a single objective truth exists, and looking for a more local truth; *action-based*, aiming for change as a direct result of the research; and *pragmatic*, looking for specific social or business benefits as a result of the research.

For the author, the purpose of this research is to provide tools to help with software development; his experience with software developers led to a wish to find better ways of doing that job. Therefore, our approach to this research is strictly *pragmatic*. In particular this means that we use aspects of different methodological approaches as seems likely to get the most effective results. As discussed in section 1.2, our two key research approaches for this work are Grounded Theory and Appreciative Inquiry. The following sections introduce them in more detail.

2.3 Introduction to Grounded Theory

Grounded Theory (GT) is a systematic methodology to construct theory through the analysis of data. It originally developed in the US medical field, where the direct value of discoveries about human social behaviour is high. Glaser’s early works, ‘The Discovery of Grounded Theory’ and ‘Theoretical Sensitivity’ [47,48] contain a good deal of discussion of the social benefits of the process, and a polemical style against alternatives.

Within 10 years Glaser and Strauss/Corbin [101] were competing for ‘ownership’ of the technique and the two resulting ‘flavours’ of GT still remain distinct. Both approaches are *positivist* in essence, though Glaser’s aim is to `discover' a single overarching theory, while Strauss is more interested in causes and effects [75].

Later still, as the technique was adopted by European researchers, Charmaz [21] tailored it to support the *relativist* philosophy [79]. The resulting variant is now known as Constructivist GT, an approach that emphasises the researcher’s impact and the restricted applicability of any results. In accordance with our *pragmatic* philosophy, the approach used in this research is Constructivist GT.

Glaser and Strauss/Corbin’s works are strong on justifications and in some cases team approaches, but less so on practical instructions how to go about making detailed choices in following the method [45]. However software engineering researchers now have access to a range of work filling this gap. A good starting point is Hoda et al.’s set of patterns instructing a novice how to set about a GT research project [54]. Other work in this area extends basic GT with detailed advice, such as Adolph's ‘Lessons Learned’ [3] and Allan's ‘Critique of Using Grounded Theory’ [6]. Finally Stol et al.’s ‘Grounded Theory in Software Engineering Research’ [100] provides a detailed and very explicit set of instructions on how to achieve academic rigour.

2.4 A Brief Overview of Grounded Theory

Traditional science assumes that theories are generated as hypotheses by the researcher, which are then repeatedly tested against reality [83]. The concept is that random theories are winnowed by the scientific process to leave only those which match observable and testable fact. Grounded Theory attempts to make theory generation into a more dependable process, based on textual analysis. Rigorous testing of the theories generated is expected to happen via other approaches.

The textual analysis is of everything relevant that is available to the researcher. Thus it might include interview transcripts, survey comments, relevant research literature, field notes from observation and anything else that can be reduced to text form. This is summed up in the GT principle *all is data***.**

The process is iterative, with analysis of initial findings from interviews or similar typically leading to changes in the research thrust and direction, and with every code written being matched against all the others, a technique called the *constant comparative method*.

2.5 Grounded Theory Step-by-Step

Figure 1 shows the techniques that we use in the Grounded Theory process.

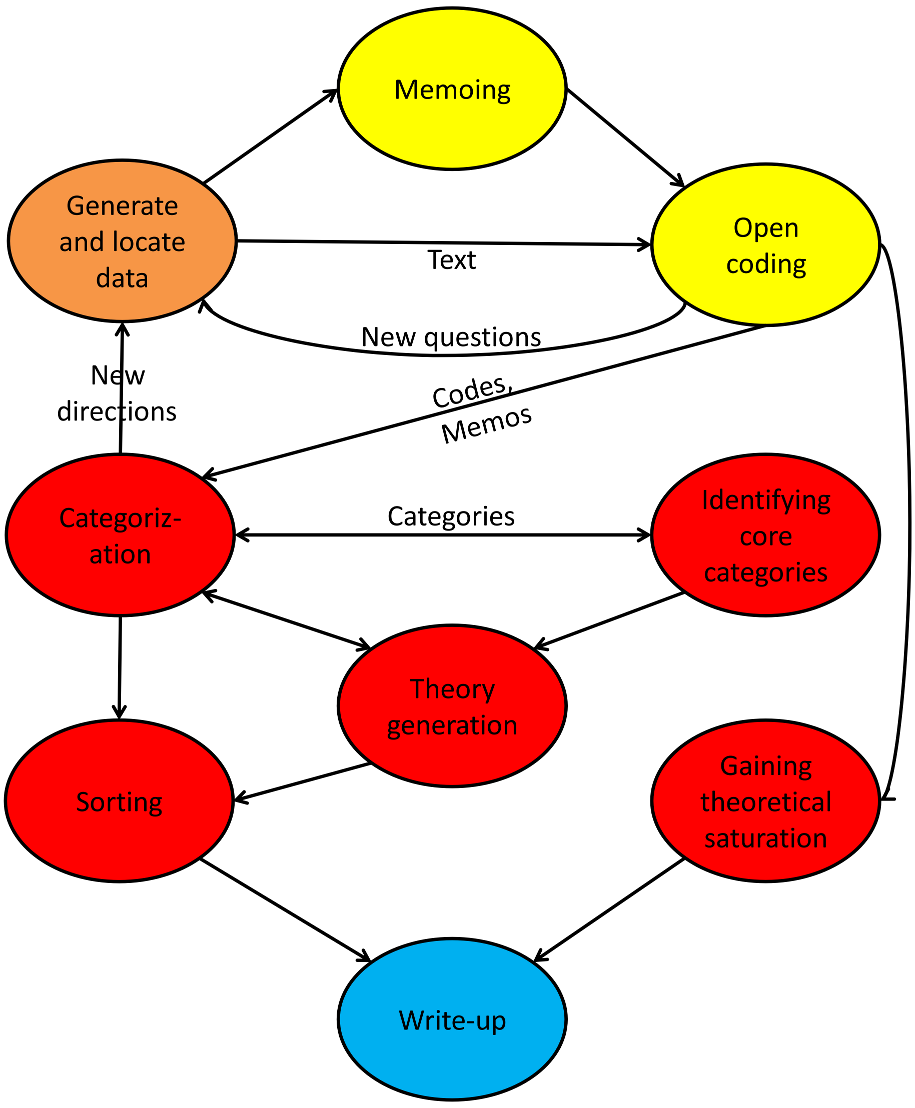


Figure 1: Grounded Theory process

Table 1 describes each technique in more detail, as follows:

Table 1: The steps in Grounded Theory

Technique	Description
Open coding	We scan each text line-by-line, highlighting points of interest. We then 'code' each to represent specific concepts. We chose the codes to be similar across documents, so that a given code represents the same concept throughout the research.
Memoing	As we do that, naturally, ideas will occur to us and thoughts

	about how the terms may be interrelated. We write these in separate texts called memos. In doing this, we are open to new ideas and concepts that may change and affect our future gathering of data. For example, if we see concepts emerging, we may explore these in more detail in future interviews.
Categorisation	Gradually, as we assign codes, we will naturally see them appear in groups of related concepts. We name these groups 'categories'.
Identifying core categories	In traditional Glassarian research, the aim is to find a single overarching Core Category: the one which covers the most interesting features in the data. Researchers are encouraged to look for categories that cover as much of the variation in the data as possible. However in this research, we are looking for not one category, but a number of concepts which can be identified and named separately; we use the validity criteria for a Core Category to identify each such concept.
Gaining Theoretical Saturation	The data gathering is considered complete when further data received does not lead to significant new concepts. This is termed 'theoretical saturation'.
Theory generation	In building a theory, we are looking for relations between concepts and categories that explain the relations between concepts and categories.
Sorting	To provide a coherent output, we need a narrative. Strauss in particular talks a good deal about the literary aspects of the narrative, in particular 'grab', the relevance to the reader [4]. The sorting process is arranging the codes, memos and categories in such a way as to produce a convincing narrative.

Write-up	Here we write up the narrative as a coherent report. We use extensive anonymised quotations from the data sources to provide rigour, and use illustrations and where appropriate to convey the information clearly.

2.6 The Use of Literature Surveys

Hoda et al. [54] recommends using existing literature in a different way from other forms of research. Grounded Theory has a major concern about being biased by existing thinking. So GT’s original recommendation was to leave any literature survey until after the main bulk of coding and ideas generation [3]. Hoda et al. disagree with this, pointing out that literature surveys are often needed due to academic pressures and the need for the researcher to be up to speed with the subject terminology. They therefore recommend a short literature survey to begin with, and a longer one once the majority of data has been coded. The longer survey may itself contribute to the coding and memo generation. However we have not seen a suggestion that written papers are coded with the same level of detail that is given to other research data.

Others suggest a similar approach: Allan [6] used a literature survey in advance of GT work to identify if there were compelling theories already in existence.

We have therefore taken the approach of an initial literature survey to learn nomenclature and avoid `reinventing the wheel'. A final, post-research survey added further detail in the context of the discoveries from our interviews. In writing the thesis, we combined the results of both surveys in chapters 3 and 4.

2.7 Incorporating Appreciative Inquiry

A further important question is how to structure the interviews to get the most helpful results. Our major concern in discussing security with experts was the danger of a litany of complaints and major problems they had seen. Much security literature, especially of the ‘black hat’ variety, amounts to this. We wanted instead to avoid the details of problems, and to focus on what had actually worked well for our interviewees.

This led us to look at Appreciative Inquiry. This method is primarily used as an Active Research method, with its purpose to change the participants’ behaviour for the better as they reflect on their answers to questions. In this research we did not need to change the participants (though we believe the reflection involved with answering our questions was beneficial, and in two cases the interviewees followed up with emails that said as much). Instead we want to find the means to help change *people like the interviewees or those they work with* in future – programmers whom we help to learn app security – and so the Appreciative Inquiry method offers a valuable contribution.

The full Appreciative Inquiry method [28] involves a cycle of four processes, as shown below.

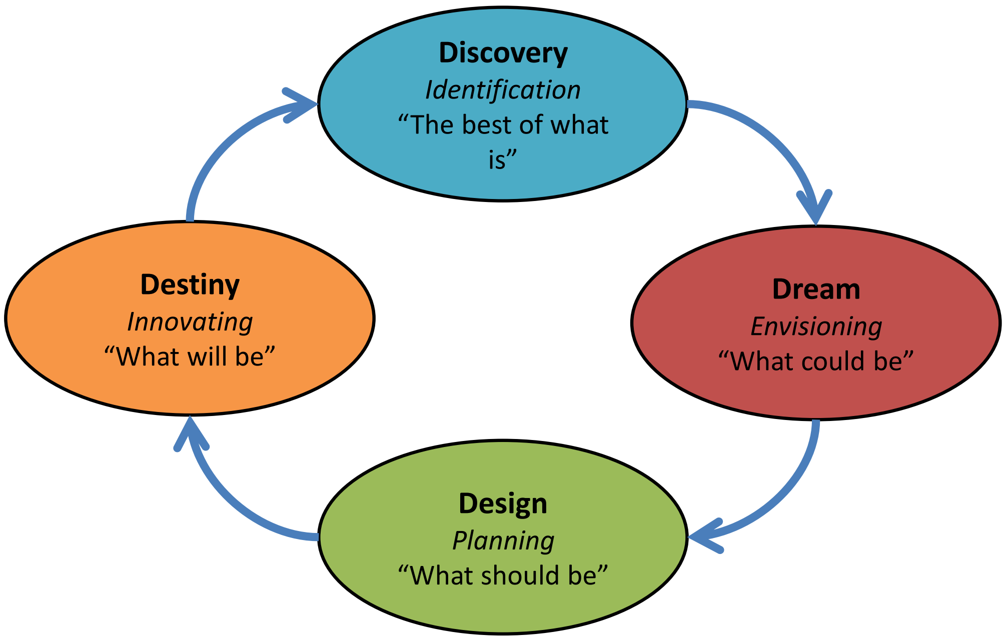


Figure 2: Appreciative Inquiry

The Discovery Phase typically concentrates on the positive aspects of the current situation, encouraging participants to visualise what has worked, and what is now working – hence the name 'Appreciative'. The Dream Phase also stresses the positive, with participants working to establish a shared vision of the future. The Design and Destiny phases then continue to produce a plan for future change that has buy-in from the participants.

Governing the use of the four phases is a set of five principles, which we may summarise simply as shown in Table 2 below.

Table 2: Principles of Appreciative Inquiry

Principle	Summary
Constructionist	Our beliefs determine what we do; thought and action emerge from relationships.
Simultaneity	Enquiry changes systems
Poetic	Organisational life is made up from stories co-generated by the participants
Anticipatory	What we do today is guided by our image of the future.
Positive	Positive emotions are needed to generate sustainable change.

Appreciative Inquiry has been effective in creating organisational change in a large variety of different organisations [27]. To use it as a research tool requires some changes that are explored in detail by Reed [89]. In this research, however, we are using aspects of only the first two phases. Each interview concentrated on Discovery: "What do you do and have you done in the past that was successful and Dream: "What would you like to see in an ideal world?"

2.8 Research Design

Our main data collection was via semi-structured interviews with software development specialists. We chose these opportunistically, mainly through their introductions from former colleagues of the researcher.

It is important to distinguish between the experts interviewed ('interviewees') and the subjects being discussed ('programmers'). The experts chosen were not at all typical of app developers; some were not developers at all, and all had more experience working with software projects than the developer average of six years [118]. However the interviews suggest that between them the chosen interviewees had worked with a typical range of programmers.

Time and practicality limited the number of interviews to a dozen. Guest [51] suggests that further interviews would be unlikely to generate much in the way of further new theory. More important is, of course, whether this achieved 'theoretical saturation'; we shall revisit this question in section 8.5. We started with an initial 'pacing interview', analysed that and used the results to direct future interviews.

Table 3 below gives an overview of the experts interviewed. For each, we have given an indication of the nature of the companies they are currently involved with and their typical role. In two cases we interviewed a second person in a different role in the same organisation, so the table shows 'organisation IDs' (OID). Some were contractors working for more than one organisation; these are shown with an asterisk in the OID column and for them the table gives the organisation for which they are currently doing the most work.

Table 3: The experts interviewed

ID	OID	Organisation type	Typical role
P1	*	Bespoke app developer	Developing apps for business clients
P2	O1	Mobile phone manufacturer	Leader of large team specialising in security
P3	O2	Operating system supplier	Developer of user-facing web services
P4	O3	Smart card specialists	Design and implementation of smart card software
P5	O4	Security-related software-as-service supplier	Architecting and promoting a secure service
P6	O5 *	Promoting industry	App security consultancy
P7	O1	Mobile phone manufacturer	Developer and software architect for OS services
P8	O6	Telecoms service provider	Architecting mobile phone services
P9	O7 *	Bank	Analysis, design and implementing changes to web-based services
P10	O8	Secure app technology provider	Architecting and promoting app technologies
P11	O2	Operating system supplier	Designing and promoting security enhancements
P12	*	Bespoke app developer	Developing apps for business clients

All had more than 30 years' experience in software development, excepting P7 who has around 20 years. All but P1 had at least 5 years' experience working with secure software development. Regrettably in terms of diversity but typically of their roles in this industry [29], all were male.

2.9 About the Author

Constructivist Grounded Theory is a relativist approach, as discussed in section 2.2. It considers the researcher to be part of the system being analysed. This section, therefore, introduces the author as a context for the findings.

Charles Weir has experience of over 30 years in commercial software development. He has worked as a programmer within a large company; as a consultant training and assisting in a large and varied number of projects; and more recently has run a software development company for 15 years producing innovative bespoke software for mobile phones.

A constant theme for his work over the last 25 years has been finding ways to improve software development. This led to early work on software testing in the 1980s, involvement in the patterns movement in the 1990s including the authorship of a book [78], and the introduction of agile development techniques in both his own company and others.

In the last few years he has led projects implementing payments and mobile ticketing on mobile phones. Like most applications software developers he had not previously been involved in software security, and had indeed avoided such projects because of the perceived commercial risk. The learning process for software security proved painful, which is what motivated this research.

The research itself led to some surprises for him. Particular was the finding that most app developers had no interest in security (section 5.1). This was because he had always been in a situation where it was obvious that unless he got app security right there would be bad consequences. Specifically, his company was developing apps for a telecommunications giant, subject to very stringent and tightly worded contracts. If they messed up they would be sued, and though the company's professional indemnity insurance would cover any penalties, the time cost of defending such a lawsuit and the resulting cost of future Professional Indemnity Insurance would probably destroy the company and put everyone out of work. In short, he was scared of the implications, and put a good deal of effort into ensuring they took a professional and responsible approach to app security.

Even so, he appreciated that although he knew about that threat, his fellow programmers would not have realised it if he had not pointed it out. After all, the contractual obligations and chain of future consequences were not part of their daily programming job, and many of them would not have been aware of them. Therefore, though he personally was very concerned when he first started developing apps that had a security requirement, it required communication to extend that concern to his colleagues.

The second surprise was the wide range of applicability of app security and privacy questions. To him, one of the first things in a new project was to identify the non-functional requirements. Before he started the research he had mentally divided apps (and indeed software generally) into those with security requirements and those without. For the ones without, he had believed that issues like security and privacy were generally not of interest to app developers. In the course of this research, he learned from discussions with academic professionals and security consultants that any apps that hold keys or passwords, use HTTPs to communicate, hold personal information, send emails, or even just read or write the file system, are subject to implied security and privacy requirements. The finding that most of the developers involved in programming such apps do not appreciate the issues is very concerning.

2.10 Introduction to the Interviewees

We shall be working with these interviewees for the rest of this thesis, so it is worth providing a more personal introduction. The illustrations in Figure 3 introduce each more personally, giving each a comment chosen to reflect a flavour of the discussion.

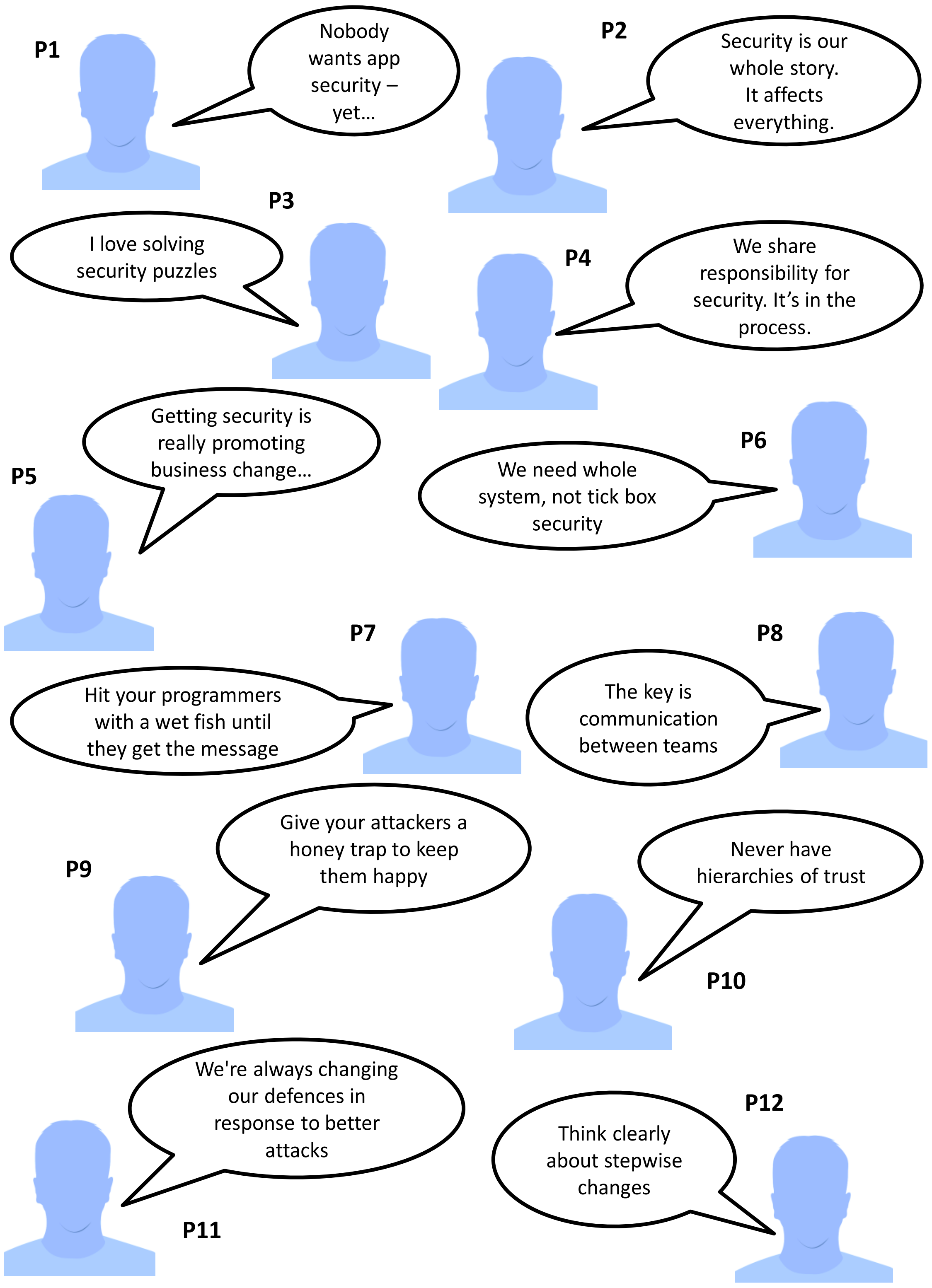

P1
Nobody wants app security – yet…
P2
Security is our whole story. It affects everything.
P3
I love solving security puzzles
P4
We share responsibility for security. It's in the process.
P5
Getting security is really promoting business change…
P6
We need whole system, not tick box security
P7
Hit your programmers with a wet fish until they get the message
P8
The key is communication between teams
P9
Give your attackers a honey trap to keep them happy
P10
Never have hierarchies of trust
P11
We're always changing our defences in response to better attacks
P12
Think clearly about stepwise changes

Figure 3: An introduction to the interviewees

2.11 Introduction to the Organisations

All of the interviewees had one organisation for which they were predominantly working. In some cases this organisation was probably their only significant recent experience (P2, P3, P7, P11); others work sometimes or regularly with a variety of other organisations. In the case of the contractors (P1, P12) their relationship with their main organisation was for a set period, typically months. The others had been in their current roles for at least two years.

Table 4 shows the organisations involved. It shows an indication of the organisation size (*small* for less than 10 staff, *medium* for less than 1000, *large* for greater than 1000, or *government* for a government department), and a subjective estimate of the organisation's position on a 'secure software capability maturity model', such as ISO/IEC 21827:2008 [56]. The organisations varied enormously, from global giants (O1) to companies with less than a dozen (O3, O8). In the case of solo programmers and smaller companies, we found the interviewees' work generally involved relationships with much larger organisations too.

Table 4: The organisations represented

People	Org	Org. size	Organisation type	Est. CMM
P2, P7	O1	Medium	Mobile phone manufacturer	High
P3, P11	O2	Large	Operating system supplier	Very high
P4	O3	Small	Smart card specialists	Medium
P5	O4	Medium	Security-related software-as-service supplier	High
P6	O5	Government	Promoting industry	Low
P8	O6	Large	Telecoms service provider	Medium
P9	O7	Large	Bank	Medium
P10	O8	Small	Secure app technology provider	Medium
P1, P12		Solo	Bespoke app developer	Low

2.12 The Interviews

Figure 4 shows the questions used as a basis for the interviews. The questions are designed to be open ended, to encourage the interviewee to cover topics from their own points of view. Drawing on Appreciative Inquiry, they focus on the positive, on things that have worked and on techniques that help to achieve positive outcomes.

Interview Schedule: Better Approaches to Secure User-facing Software

Introduction – establish context

- Please could you tell me something of your own background?
- What is your current role, and what do you find yourself doing day-to-day?
- How did you first get involved with developing secure software?

Exploration

- Can you think of a particular triumph in your work? How did you achieve the security aspects of that?
- Please could you give examples of a secure system that has gone well? Or not gone well and been fixed?
- How did you initially learn about adding security to software development?
- How do you learn more now?
- What aspects of your team (your work) made them particularly good at secure software?
- What is the most successful technique you have found?
- What advantages are there in the development of the app end (browser or mobile) over the server end?

Clarification

You mentioned [specific technique]. Can you tell me a little more about that?

Figure 4: Interview schedule

In many cases interviewees covered answers to later questions in the responses to earlier ones, or even before any questions were asked. The clarification questions were not appropriate to all the interviews, so were sometimes omitted. In all cases except P7 the discussion around the questions took at least an hour and in some cases (P4, P5, P8 and P9) more than two hours.

2.13 Analysis of the Interviews

Some of the interview questions related to the experts' analysis of how to achieve secure app development; others to their own history and ways of learning about secure app development. Thus we can distinguish two forms of information from the interviews: information about how the app security experts had achieved their expertise and kept themselves updated, and information they had about best learning approaches for those working with them. Since most of the interview content was about this second topic, treating the interviewees as peers to be consulted rather than subjects to be analysed, in this thesis we refer to those involved as 'interviewees' or 'experts'.

In this thesis we quote extensively from the interviews. To convey correctly the context and protect the confidentiality of the interviewees, we have amended the quotations appropriately: names are changed; square brackets show additions and replacements; ellipses show removals.

3 Existing Research on App Programmers and Security

This chapter examines the literature around the main topic of the thesis: what techniques and ideas will appeal to development teams and lead to them developing more secure app software. Thus it considers work on programmer motivation, learning and improvement.

3.1 Two Types of Literature

Recall the overall research question for this thesis:

> *RQ1 What techniques and ideas will appeal to development teams and lead to them developing more secure app software?*

We found in researching literature that the existing work on the subject falls into two different categories: there is literature *about* programmers, and there is literature *for* programmers. We realised we need to analyse the two types of literature differently. In this chapter, therefore, we analyse literature *about* programmers in terms of its contribution to our research question. In chapter 4 we shall examine literature *for* programmers in terms of its usefulness in helping programmer learning.

3.2 Focus of Existing Research

As discussed in section 1.1, existing research on the security of software development focuses on three aspects:

- Programmer motivation,
- How do programmers learn, and
- What do programmers need to learn?

The following sections explore each in turn. Throughout this section, we evaluate each work in terms of the impact and relevance it may have on the research questions we identified in section 1.2. In line with our pragmatic approach to the research, we comment on each only in respect of its relevance to those questions.

Clearly while app development has its own environments and cultures, many of the problems are similar to those in other domains, especially Web UI development, and desktop application development, and to a lesser extent server development. We have found literature about programmer motivation relevant only to programmers in general, and similarly much of the work on how programmers learn is general. There is more work available specific to app developers on what they need to learn.

Two closely related domains are Web UI software development and workstation software development. Certainly the study of software security for Web UIs is more mature, and there has been a good deal of work on specific kinds of security issue, such as cross-site scripting, that apply to that area. However that is not helpful to this research. On the more general issues of how to get development teams to deliver better software, we have found almost no research specific to Web UI or workstation software.

3.3 Programmer Motivation

This and the next section explore writings related to the research question:

> *RQ2 What motivated the experts themselves to learn software security; how did they do so; and how do they continue to learn?*

There is a good deal of literature on programmer motivation. Beecham et al.'s survey in 2008 [15] found some 92 papers on the subject. However virtually all of the research cited is about *motivation to do the job* of programming, rather than *motivation to change behaviour*. There is an implicit assumption that a well-motivated programmer will do a 'good' job. Tom DeMarco's popular book on the subject, 'Peopleware' [33], makes the same assumption.

In this work, we are interested in what makes a well-motivated developer choose a new form of behaviour: learning about software security.

What Beecham et al.'s survey does in particular is to establish the kinds of things that motivate programmers. Specifically, they identified that professional programmers tend to be motivated most by:

- Problem solving,
- Working to benefit others, and
- Technical challenge.

Interestingly, a fear of failure is not among the list of motivators. So the classic security motivation of 'a terrible thing might happen' will not tend to have much impact in encouraging programmers to learn about security. This agrees with the findings of Xie et al. [112], whose survey interviewed programmers to investigate why they believed they made security errors; they found a consistent tendency to treat security as 'someone else's problem'. Though the results are limited in scope to highly experienced developers in US companies, it seems reasonable to conclude that the conclusions may apply more widely.

We can conclude that to inspire programmers to become adept at software security will require emphasis on the benefits to others and the technical challenge of the problems involved.

3.4 How Do Programmers Learn?

There is relatively little literature on how programmers learn, whether about novices or professional programmers. Johnson and Senges [59] studied how programmers learned to function in a complicated organisation, Google. They concluded that the majority of programmer learning there was peer learning, facilitated by strong corporate standards and culture; obviously the results are limited to that organisation. Other studies have incorporated the concepts of programmer learning into the wider term of Software

Process Improvement (SPI). So, for example, a wide-ranging quantitative study by Dyba [36] examines learning as one aspect of SPI; it differentiates *Exploitation*, the dissemination of existing knowledge, from *Exploration*, the gaining of new knowledge; it concludes that both have a positive effect on productivity but does not explore mechanisms.

A little-known work by Enes and Conradi [40] used interviews to discover how professionals, including programmers, acquire their expert knowledge. It concludes that the preferred learning mechanisms are all informal ones: especially on-the job training and personal interaction. It also highlights, as an important factor, professional pride in having 'expert areas' of competence. Unfortunately for our purposes, the results include only a very small sample of programmers (4) in a limited area (Trondheim).

Murphy-Hill et al. explored how developers find new software tools [72]; the research was more wide-ranging and included both an initial survey and a larger scale, 79 participant, diary-based survey. They concluded the event is relatively rare, and happens more through joint working than recommendations, though the only solution proposed was to create a tool recommendation website. Proksch et al. used an extensive and carefully analysed literature survey to explore such a recommendation site in detail [85]; the result reads as a requirements specification. So far there is little evidence of such systems being trialled in practice.

In the context of learning about software security, a particularly important finding is that of Conradi and Dyba [26]. Based on interviews with 21 development team members in 5 Norwegian companies, they analysed carefully the effectiveness of written routines on software process improvement. They identified that programmers have difficulty with learning from the output of process improvers, and particularly with learning from formal written routines. As a result, developers tend to avoid such approaches:

> *Developers are rather sceptical at using written routines, while quality and technical managers are taking this for granted. This is an explosive combination. (Conradi and Dyba* [26]*)*

This suggests an 'impedance mismatch' between those who write instructions and processes for developers, and the developers themselves who are expected to carry them out. While the geographical scope of the research gives it limited external validity, anecdotal evidence and personal experience of the authors tends to confirm the finding in respect of UK and US developers. We can speculate that security experts tend to think in terms of complete lists of issues and ways to break software; developers think in terms of simplest ways to create desired functionality.

Thus though one might expect the most effective approach to teaching security to be a prescriptive set of instructions to programmers what to do, a ‘Secure Development Lifecycle’ such as those promoted by Microsoft [69] and others, in practice those do not appeal to developers. This conforms to the author’s own experience; he has never encountered an app development team voluntarily adopting a process of that kind. This means that for app developers we need to find a more lightweight, less prescriptive approach.

A different approach to teaching programmers was the ‘patterns’ movement, discussed in more detail in section 4.2. Yskout et al. tested experimentally the effect of using security patterns in server design; though the paper states there was no benefit, in fact the results suggest a benefit but were statistically inconclusive [116].

Recently several teams have investigated how app developers learn security. Balebako et al. interviewed a dozen app developers in small to medium sized US companies, and surveyed over 200 US app developers to find out their approach to security and privacy issues. The survey was carefully constructed to avoid bias, and concluded that most developers approach these issues using web search, or by consulting peers [10]. Interestingly they also found that security and privacy behaviours were only weakly correlated.

A survey by Acar et al. [2] also concluded through a survey of nearly 300 successful app developers worldwide that they learned security through web search and peers. They went on to use a well-crafted practical experiment with over 50 Android developers to evaluate the effectiveness of the different ways of learning app security; this produced the surprising result that programmers using digital books achieved better security than those using web search.

Two projects, by Xie et al. and Nguyen et al., have developed IDE-based tools to teach programmers by detecting possible security flaws in Android developments [77,111]. The approach is promising, but obviously requires programmers to adopt the tools; also Xie et al.’s project is now finished and Nguyen’s has not yet reached the stage where its effectiveness can be tested. Others, such as Near and Jackson, and Lerch et al. [65,76], have code analysis tools to detect security defects; these work but provide only limited feedback to developers. So far we are not aware of any literature analysing the effectiveness of such approaches.

3.5 What Do Programmers Need to Learn?

In exploring the next research question,

RQ3 What are the most effective techniques to deliver app security?

we may be able to establish what kinds of learning are required from looking at the security issues found in the past work of programmers.

Enck’s ‘Study of Android Application Security’ [39] used binary code analysis of a large number of free Android applications. This approach is limited to the kinds of error that can be found with such analysis. Thus they found misuse of APIs, such as using cryptographic APIs in ways that reduced the security provided by the cryptography; and they found access to privacy-sensitive APIs, such as APIs that gave personal information about the user. They could not find errors such as insecure storage of credentials (in publicly accessible files, for example) or misuse of insecure network connections (such as using HTTP where HTTPS was needed). Fahl’s analysis of Android SSL insecurity [42] used a similar approach to analyse apps’ use of secure web connections; again they are limited to detecting inappropriate use of SSL APIs, which they found in 8% of apps. Certainly we can conclude from both papers that developers need to be mindful of careful API usage.

A recent paper by Nadi et al. [74] uses surveys of programmers and of their comments on discussion pages to examine the reasons why developers had issues using these SSL APIs. They were asking developers who had had trouble, which for our purposes might be regarded as sample bias. Unsurprisingly the conclusion could best be summarised as ‘because the APIs are difficult to use’, with the recommended solution being to document them or to wrap them with libraries or code generation tools; it appears from this paper too that developers do not see education on security as something they need.

Vidas et al.'s paper 'All Your Droid Are Belong To Us' [105] takes a different approach, and examines attacks on Android phones. This is not in the context of app security, but rather in the context of the whole phone, where apps are seen as potential malware and the solutions are in terms of OS improvements. Acar et al.'s recent literature survey of Android app security [1] provides a comprehensive view of the state of the art, concluding that improvements to APIs and the use of web technologies are helping; however they do not otherwise address the expertise of developers.

The openness of Android encourages such studies, but iOS gave rise to a smaller number of similar evaluations such as that of Dai Zovi [31]. Unfortunately for our purposes this reads more as a summary of the security facilities provided by Apple rather than an objective and critical assessment of risks to app developers.

4 Learning Resources for Programmers

This chapter continues the literature survey by examining learning resources available to programmers. It explores three different movements that have led to the creation of such literature: the security patterns movement, the practitioners' movement and black hat literature. It then considers other resources available: web-based static sites and video. It uses rankings to indicate which are most used by developers.

4.1 Introduction

Our final research question in section 1.2 asked:

> *RQ4 How should we effectively introduce security to app development teams?*

To address this we need resources to help programmers to learn. Whereas chapter 3 explores research on how programmers learn; this chapter looks at the resources available on good practice that may appeal to and benefit programmers.

We identified several categories of such resource as follows:

1. Static literature such as books, e-books and papers;
2. Websites, blogs, and bulletin boards; and
3. Online video

To evaluate the resources, we need to consider what attributes we need from them. Since we are looking for learning tools, academic validity though relevant must be matched against considerations such as ease of access, readability, approachability and didactic approach. For example, academic papers that are not freely available on the web are unlikely to be widely used by non-academic programmers, as are books that are out of print. Similarly books that sacrifice readability for academic rigour or completeness may be forced on students doing academic courses, but are unlikely to be widely read by developers if they have the choice of more readable alternatives.

There are many excellent books describing the theory and practice of software security, such as Gollman's 'Computer Security' [49], Pfleeger 'Security in Computing' [82], Schneier 'Secrets and Lies' [93] and Anderson 'Security Engineering' [7] ; these work at a level that is not helpful as anything but background reference for a software developer. Surprisingly few books are targeted at software engineers. Three separate approaches appear to have led to such material: the security patterns approach, the security practitioners' approach, and the black hat approach. We explore each of these in the following sections.

4.2 Security Patterns

Security patterns are a development from the software design patterns movement first brought into prominence with the 'Design Patterns' book [46], a very popular book with programmers. This was in turn built on the architectural patterns work by Christopher Alexander [5]. An important feature of patterns is that they divide up a problem domain and provide multiple positive solutions. A series of patterns may provide a variety of competing solutions for the same problem, and the patterns will provide information ('forces') to help practitioners decide which one to use. 'Design Patterns' describes a set of ways to structure object-oriented code, mainly to achieve good partitioning between components. The term 'Security patterns' extended this idea to mean 'Design patterns associated with software security'.

The security patterns literature shows strong signs of its position as the meeting of two cultures. The patterns movement authors emphasise the positive aspects of their work ('solutions') and single out particular aspects of a domain; the security movement authors emphasise restrictions ('threats') and aim for complete coverage of a domain.

The earliest security patterns paper ‘Architectural Patterns… ‘ was written by design patterns experts Yoder and Barcalaw [113], and is designed to be approachable to programmers. Though the patterns are straightforward and high level, the detail in each is quite complicated to follow. ‘Security Design Patterns, part 1’ by Romanovski [91] is another early attempt to distil security patterns. It reads as fairly naïve, without the authority and references one might expect.

Kienzl et al.’s ‘Security Patterns Repository’ [63] describes some 20 ‘structural patterns’, with noun names like ‘trusted proxy’; and some 10 ‘procedural patterns’, with verb names like ‘document the security goals’. Though some are ‘mini-patterns’ only sketchily described, most are valid and useful explorations of security techniques for app developers; and each includes an examination of the impact on different aspects of security and performance. A version is publicly available.

Of the two best-known security patterns books, ‘Core Security Patterns’ [98] attempts to cover the entire domain of Java Enterprise security, but is now somewhat out of date; naturally it avoids dealing with application software. The book ‘Security Patterns…’ by Schumacher et al. [94] incorporates work by several authors and teams, covering security analysis through to implementation, though it too has little on application security. Unfortunately the early chapters read as a ‘secure process’, with the same issues as highlighted by Conradi and Dyba (see section 3.4). Bejtlich, a respected industry expert, included ‘Security Patterns’ in a comparative review [16] but found it less valuable in practice than other books discussed in the next section.

How valid are security patterns as learning material for programmers? Clearly there is bound to be overlap: surveys of the literature written between 1997 and 2005, the main days of the patterns movement, claimed variously only 179 distinct patterns [52] or only 36 ‘true’ patterns [114] out of the several hundred available. Yskout et al. trialled a controlled experiment to see if developers benefitted from using security patterns [116]; the results suggested a benefit, but were statistically inconclusive.

Interestingly, there has been surprisingly little effort since that time to add to the canon of software security patterns. Much recent pattern-related literature in the security field has used the name ‘pattern’ in its wider sense, referring to the categorisation of similarities across different items (such as attacks, or text for analysis), rather than ‘security design patterns' as tools for developers and practitioners; an example is the book 'Cyberpatterns' [17].

4.3 The Practitioners' Movement

Security practitioners' literature for programmers comes from people studying software security, who offer their knowledge in a form suitable for programmers. The most approachable we encountered is Gary McGraw's 'Software Security' [68], which both sets forward a clear approach, and uses a fairly readable style [16]; however it is structured as a secure development process, which limits its appeal to developers (see section 3.4).

The most useful learning books for software developers are those that convey information in a relatively terse and readable form, and in manageable chunks. For this Howard, LeBlanc and Viega's '24 Deadly Sins of Software Security' [55] provides a good introduction; its format is similar to that of the patterns literature.

An alternative resource is books targeted specifically at particular platforms. For Android, there are books explaining the security model and development techniques; examples include 'Pro Android 4' [64] and 'Android Security Internals' [38]. The first provides general security techniques and code in one chapter; the second provides a more complete overview of the Android security system; both provide examples. For iOS there are equivalents, such as 'Learning iOS Security' [11]; though this is more a description of security features than a guide to avoiding security issues.

On the web, Apple has web pages to help developers learn the details of app security for iOS [8]; Google provides a rather more rudimentary set of hints for Android [50]. All of these platform-specific resources consider only code-level security and do not emphasise the 'whole system security' aspects of app development.

4.4 Black Hat Literature

Some of the most popular (see 4.5) platform-specific security books are the ones with a 'Black Hat', attacker, approach. For example the Android Hacker's Handbook [35], and its corresponding versions for iOS and web apps, contain a good deal about exploits against the operating system, a certain amount about analysing existing apps, but little about how to guard against exploits as a developer. The split into chapters each examining an area with a consistent format is reminiscent of the patterns literature. Further books in the same series include versions for Web Applications, for Mobile Applications, and for iOS applications. Chell's Mobile Hacker's Handbook [22] takes a similar approach, covering iOS, Android and even Blackberry platforms, and does provide limited advice for developers.

There are a number of papers exploring weaknesses in mobile apps, especially Android, and possible reasons for them. For example Egele et al. studied the misuse of cryptographic APIs [37], concluding that it was widespread and required better APIs; Fahl et al. [42] studied SSL use, concluded some apps were vulnerable to man-in-the-middle attacks, and suggested ways to solve the problems; however the academic format doesn't appeal to most developers.

4.5 What Books Do Programmers Use?

We can gain some idea of the popularity of books from their sales on an international bookseller such as Amazon. Chevalier has established that this correlates well with actual sales [23]. We should of course be wary of making deductions from these figures: we do not know the proportion of purchasers who are programmers; we do not know how many readers share books via libraries or other ways; and we do not know what fraction of purchasers do not read the books. Table 5, however, shows selected rankings.

Table 5: Amazon rankings for selected books at January 2016

Book and reference	Amazon bestseller rank ('000) (Low numbers mean popular)
Design Patterns, Gamma [46]	24
Android Hacker's Handbook [35]	86
Security Engineering, Anderson [7]	120
Secrets and Lies, Schneier [93]	141
Android Security Cookbook [67]	512

Software Security, McGraw [68]	627
Learning iOS Security [11]	900
Security Patterns [94]	918
Application Security for the Android Platform [96]	1218
Core Security Patterns [98]	1466

From this we can see that the most popular security books are indeed those that take a ‘black hat’ approach in a specific domain, followed by the general overview books. McGraw’s book, and the more ‘white hat’ approaches to software development sell relatively badly, and the security patterns books sell hardly at all. By comparison with the sales of the ‘Design Patterns’ book we can conclude that only the black hat, ‘Android Hacker’s handbook’, approaches being a book that might be expected to be found on many bookshelves.

This correlates with the finding of section 5.1 that few programmers are motivated to find out about software security; it also correlates with the finding of section 3.4 that secure development processes such as those of McGraw’s book do not appeal to software developers.

4.6 Web-Based Information Sources

Software security is a fast-moving area, and website-based sources have the advantage of being easy to update, and easy to build progressively as resources allow. They have a further advantage: web links are easy to insert into discussions or answers on the most popular programmer sites such as Stack Overflow.

The classic application developer site on application security is Microsoft’s [70]. It covers a range of topics ranging from Microsoft’s complete secure development life-cycle, to implementation details for securing Microsoft products. It has the disadvantage of stressing a process-based approach, the Microsoft Secure Development Process, which may limit its appeal to developers (see section 3.4). Apple has web pages to help developers learn the details of app security for iOS [8]; Google provides a rather more rudimentary set of hints for Android [50]. Both are valuable, but strictly technical – they consider only coding aspects and ignore ‘whole system security’ issues.

The Open Web Application Security Project (OWASP) community-written ‘Developer Guide’ [119] has a great deal of content, but is difficult to access in a manageable form; all the information is in Markdown format, and there do not appear to be compiled PDF or e-book formats. It is unlikely that many programmers access it. Another document deriving from the OWASP source is ENISA’s ‘Smartphones Secure Development Guidelines for App Developers’ [41]. This is a tersely written list of do’s and don’ts for secure app development; it does consider wider issues than the strictly technical, but is not an easy read for those inexperienced with security.

The community-written OWASP Top Ten Mobile Risks site [120] is a widely accessed[1] resource detailing specific programming issues and how to avoid them. Its authority and availability make it very effective, though it does not consider ‘whole system security’ issues. A further commonly referenced resource is the SANS Institute site [92]. SANS is a commercial organisation supplying training as well as free information services, and does have a ‘whole system security’ approach. However SANS has little information specific to app development.

Some third parties have also constructed websites. A particularly approachable one is the Android developer security site by popular blogger Simon Judge [60]; it divides its content into a collection of separate homilies, each argued independently. Again the format is reminiscent of the patterns form. Though approachable, it has not been widely accessed by programmers [61].

App programmers tend to use web search and discussion sites as their primary source of information on security [2]. For programming questions, the two dominant sites are of course Google and Stack Overflow. There was more than a touch of truth underlying the joke suggestion that programming be renamed ‘Googling Stack Overflow’ [102]. The main programmers’ Stack Overflow site has a substantial set of discussions covering

[1] Based on Google rankings in February 2016

application security[2]. A separate Stack Overflow site is devoted specifically to security, but tends not to handle programming questions.

Unfortunately, as a learning resource, Stack Overflow and similar bulletin boards have a significant flaw: they are poor for gaining an overview to a topic, and actively discourage questions that do not have focussed answers. Requests for help on security are generally answered with references to the resources listed earlier in this section. A detailed analysis of the topics on the Stack Overflow site [12] found little in the way of overview discussions. Thus Stack Overflow is valuable in helping programmers sort out problems they know they have, but does not point out problems that they do not know they may have; most security problems are likely to be of this second type.

Discussion sites have a second problem related to security: their answers tend to be of questionable accuracy especially when they quote code. Acar et al. [2] analysed answers on Stack Overflow to app security questions, with worrying conclusions:

> *[Of 139 threads analysed] we categorised 41 threads as being on-topic… Of these, 20 threads contained code snippets. Half of the threads containing code snippets contained only insecure snippets. (Acar et al. [2])*

[2] 1400 tagged ‘Android’ and ‘Security’; 700 tagged ‘iOS’ and ‘Security’. 18 Dec 2015.

5 Motivation and Approaches to App Security

This chapter discusses findings from the Grounded Theory analysis. It examines the motivation and approaches for developers to learn about and implement app security. First we introduce the surprising discovery that few app developers are motivated to learn about security, let alone implement it. Then we examine the original motivations of the interviewees to learn app security and their approach to continued learning; we contrast their different approaches to working with others; we conclude that the discipline of app security is at an early stage of development, and suggest how it may develop.

5.1 Why Is Motivation Important?

Early in the cycle of interviews, we learned that actually most programmers have very little interest in security for mobile apps.

> *Very, very, few developers are actually interested in security (P1)*

Thus the answer to the question ‘how do programmers learn about security’ most often seemed to be ‘they don’t’. The reason for that lies in motivation; most app programmers have little motivation to work on security.

> *“You can see that from the Apps World [exhibition] where there’s no mention of security at all. It’s not on people’s radar.” (P1).*

Accordingly this chapter explores motivation: what motivates the interviewees themselves to implement security in their systems; and how best they encourage that motivation in others.

5.2 What Motivates Our Interviewees?

The experts differed widely in their original reasons for learning about software security; there was correspondingly little agreement on how best to motivate app programmers generally to produce good secure apps.

The Grounded Theory analysis of the interviews highlighted four forces motivating a programmer to learn and act on software security, as illustrated in Figure 5.

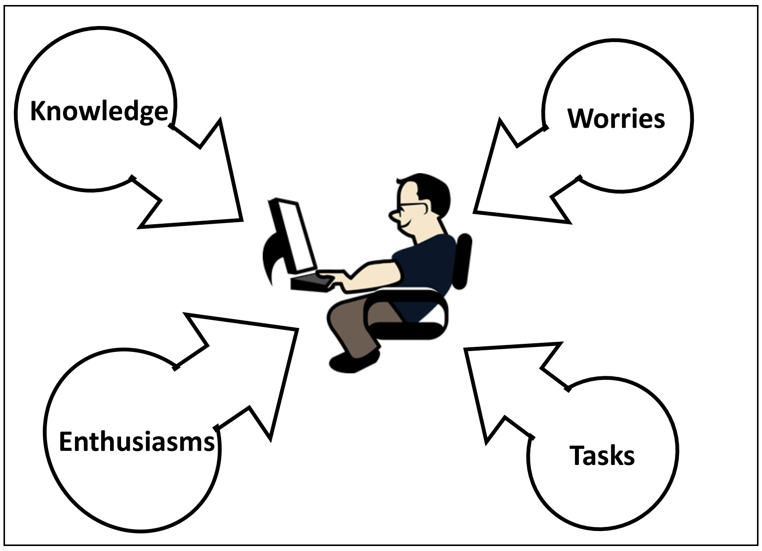


Figure 5: Motivation forces on a programmer

These forces are as follows, with some examples from the interviewees.

Knowledge: the knowledge and skills that the programmers have learned in the past or gained through experience on how to deal with software security issues.

> *I never learned from reading books; I never took any courses at college on computer security. All of my work was basically self-taught. I think that there are certain people who have a passion for information security, and those are the people… It is hard to learn academically.* (P11)

Tasks: the formal and informal assignments of code to write, changes to make, training, and related work that the programmer has as their overt job.

So, we send a couple of staff every year to the [OS Manufacturers] Conference. And secondly, [we learn from] information from our suppliers ... technical information. (P2)

Worries: the concerns and fears the programmer has about what they are doing.

[I was called out to handle a security issue] and so it was Christmas Eve I was driving down to a data centre, doing a complete factory reinstall to wipe out any traces of it, couldn't get back to my family. And I said "I never want to have this happen again, and I am going to do everything I can to make sure this never happens again" P11

Enthusiasms: the positive inspirations that motivate the programmer to make specific choices.

"Actually when I was a kid – fortunately, I never released any of this stuff – I did actually take copy protection off games for the intellectual challenge of this" (P3)

Surprisingly, we found a tension between these as two pairs of alternatives: those who saw knowledge as a motivation did not feel the need for explicit tasks and vice versa; those who felt worries were a motivation did not consider enthusiasm and vice versa. Therefore, where an expert's interview expressed a position on these forces we express that position as a location on a scale: knowledge versus tasks, and worries versus enthusiasms. For example, an expert who expressed strong views that security should be part of every relevant activity in software development would be represented at the 'knowledge' end of the scale; an expert who mildly suggested that security could be included as the tasks of penetration testing and app hardening would be placed towards the 'tasks' end of the scale.

We found similar tensions between approaches to different views on implementing security and on the role of teamwork. These tensions we also represented on a second pair of scales: teamwork versus individual rigour, and influencing versus directive approaches. A third pair of scales compares a preference for checklists versus individual rigour, and for considering the attacker versus considering stakeholders.

Thus in diagrams in the following sections 5.3 and 5.4 we position the views or information expressed by experts on specific topics against axes representing each of the two related scales. Each diagram shows only the experts who expressed a clear opinion, and shows clusters where several shared roughly the same position. The resulting pattern highlights the range of views expressed.

5.3 Reasons for Learning

Some of the interview questions were about the experts' own experience and histories (treating the interviewees as subjects – see section 2.13). The Grounded Theory analysis highlighted differences in the interviewees' own motivations for both originally learning about software security, and for continuing to learn.

5.3.1 Experts' Reasons for Learning

Figure 6 expresses the original motivations for the experts themselves for learning about software security. Where they learned 'accidentally' or while doing other things, this is shown towards the 'knowledge' end of the axis; where they learned as a specific part of their job it is a task. Similar the vertical axis shows whether they learned out of enthusiasm, or because of concerns about the potential impact.

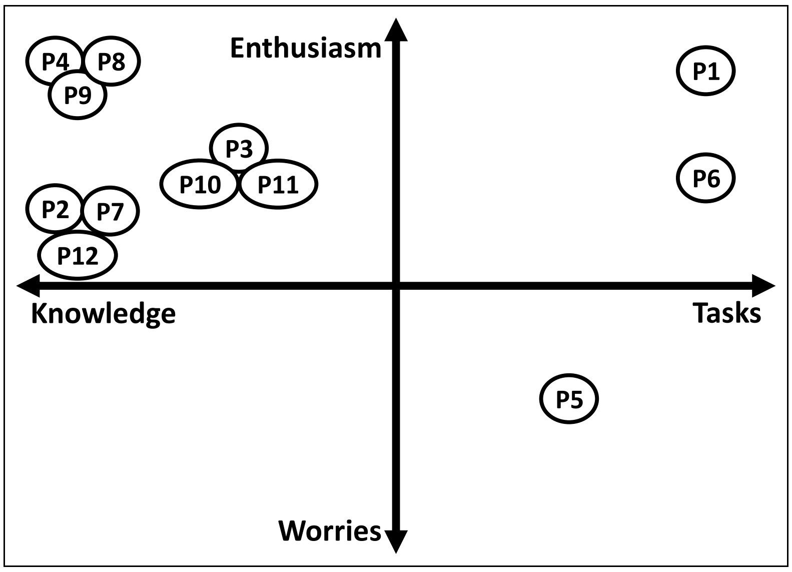


Figure 6: Reasons for original learning

The reasons varied significantly. Most had learned from day-to-day experience, given enthusiasm for the security aspects of that experience; some had started as hackers:

So the first security problem I ran into was at my dad's businesses. He was using it for accounts, and ... the accounting system had a bug ... that was affecting his balance sheets – he had a deadline! ... So we needed to find a way of making a copy of copy protected software. ... So I started off with hacking there, and it became a challenge so I then started hacking anything else that was protected! I developed quite a high sophistication in terms of disc based security, which would later be of huge benefit to me.

Others had started on projects which required security:

"[While at college] I had three very fun summers working on top secret projects and things like that, which had a fair amount of security in it." (P12)

"I did a lot of firmware work on a magnetic stripe card reader ... that had a number of security features... I definitely got the [security] bug there". (P4)

Only P1 had decided to learn about software security as a career decision – to build experience and credibility in a new area.

[I went out and] I discovered [about security]: black hats and white hats. [I] go and talk to [them], and go to conferences and see what the attackers are doing.

5.3.2 Experts' Reasons for Continued Learning

Looking at the experts' reasons for continued learning, it emerged that there was more consistency; for most it was an informal task in addition to their normal day job, and they did it on an ad-hoc basis. Only P3 and P11, who work for a global, security aware, company, received security-related training; and P1, in his role as author, assigned app security learning as part of his normal work. Most kept up to date through a background task of following appropriate internet media – Bruce Schneier's 'Crypto-Gram' email [19] was the most commonly mentioned medium (P3, P7, P8), or:

"My work screen has a Twitter feed just running up the right hand side. Whenever I get to enough of a break that I can glance over I'll take a look at whatever is currently up there." (P7)

> *"I listen to a few podcasts… Security Now… with Steve Gibson on the TWiT Network" (P12)*

Figure 7 shows the experts' reasons for their continued learning. Not all the interviews covered this topic, and thus three of the interviewees are omitted.

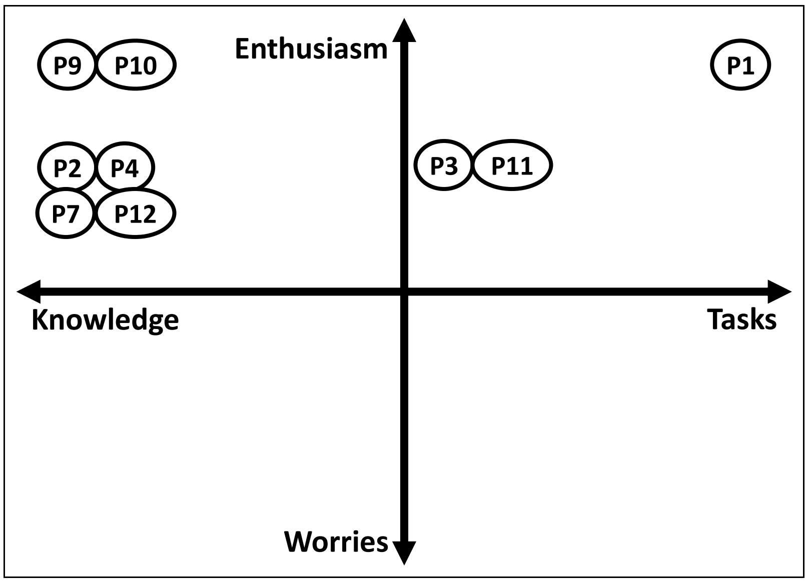


Figure 7: Motivation for continued learning

5.4 Motivating Programmers to Learn

The following sections analyse the responses of interviewees consulted as experts, rather than subjects.

The GT analysis showed a key category of the discussions to be 'how to motivate programmers'. This section explores the interviewees' views on ways of motivating programmers; it showed an unexpected disparity between their approaches.

All the interviewees who discussed programmer behaviour stressed that programmers had a tendency to avoid security issues and concentrate on delivering functionality. Some highlighted that few undergraduate level computing courses incorporate security into normal examples and practice.

> *"So for the majority of people who are currently going through various computer science degrees, security doesn't really come into it at all, in any real context". (P10)*

Many correlated general life experience, software development experience, and especially formal software development experience with ability at software security. They stressed the difficulty in motivating inexperienced developers:

> *"When I'm talking to 22 year old phenomenally brilliant mathematician software developer who has got almost no life experience at all – how do I make him care about things that seem unimportant to him?" (P5)*

However the interviewees showed little consistency in their approaches to solving this problem and motivating programmers to work on security, as follows.

5.4.1 Enthusiasm or Worry?

Some felt the motivation for security should be worry, where the impact of poor security is a threat to the programmers:

> *"We'll need a mass security event [caused by a mobile app] to get programmers to take app security seriously" (P1)*

Other saw it better as an enthusiasm, wanting programmers to be passionate about doing a good job on security:

> *"trying to talk to my developers about this and trying to come up with techniques that make them think about it in a way that makes them care about it" (P5)*

This sometimes could also be a reaction to the costs of the 'worry' approach:

> *There are too many technologists and guys with sensible shoes whose mission is to make you frightened, that will make you pay an awful lot of money, and whether you are Deloittes or Ernst and Young, or whoever, privacy impact assessment, on and on it goes. (P5)*

5.4.2 Knowledge or Task-Based?

Some represented making systems secure as part of a process, where developers do the right thing because they are expert and knowledgeable:

"So you are going to have to get developers to understand computer science, and the consequences of the code they are writing" (P6)

Others saw the adding of and planning of security as part of the functionality requirements and thus as a specific task.

"Mine is much more practical. I need it to work, I'll put something together that actually does the job, and I will learn whatever I need to learn to do that. And then move on, if necessary." (P4)

We observe that these external motivators for programmers naturally follow the same axes as the motivators the experts had had for their own learning. Figure 8 shows how the experts who expressed views are positioned on the same axes.

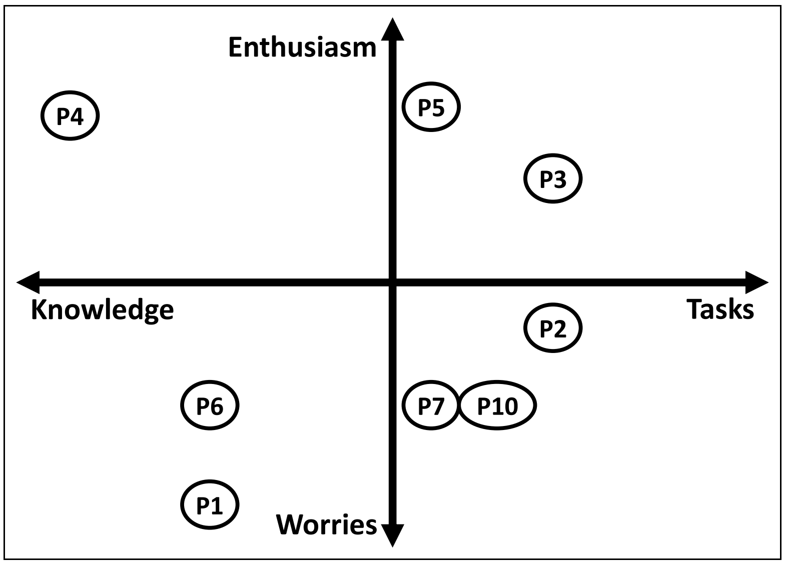


Figure 8: Recommendations how to motivate app programmers

5.5 Approach to Teamwork on Security

The GT analysis also highlighted a key topic 'Security in the development process'. Further analysis showed a key category of team approaches to achieve app security. Whilst the experts tended to agree on the importance of this, they differed on best approaches to achieve it. This section explores their approaches both to teamwork and to changing the behaviour of those teams. We found a considerable variation in approaches, and have highlighted these in a diagram similar to those in the previous sections.

5.5.1 Teamwork versus Individual Rigor

There was agreement that team attitudes are very important in creating secure software. Some experts stressed the communication between and within teams:

> *"And I think one thing that we were incredibly good at with [a specific project], is bringing the entire project team together probably with the aid of, as well as the formal meetings, some of the more casual discussions over a beer. And so everybody fully understood the scope of what everyone was bringing to the table and there was never any of the artificial formalities that sometimes you can get around these projects where it feels uncomfortable to pick the phone up to somebody." (P8)*

Others stressed individual rigor, as their primary tool. For example:

> *"I tend to look at things in a stepwise way. Certainly when you're evolving software, you don't necessarily have formal proof but you can go in sufficiently simple steps that you can see that it's obviously correct." (P12)*

We saw the latter view expressed usually related to single developer situations where there were no others with whom to discuss security.

5.5.2 Changing Behaviour - Influencing versus Directing

Another distinction that emerged is that some saw their best means for they themselves to influence the team members as directive, exerting authority:

"I had success [by] whacking them over the head with a wet fish" (P7, speaking metaphorically).

Others saw their role as influencing, questioning and encouraging:

"[I] throw out a few 'what ifs' you know, what if I did that, and get somebody who is aware and will have an understanding of what you are suggesting, and they will counter with a sensible response." (P8).

Figure 9 shows these two contrasts.

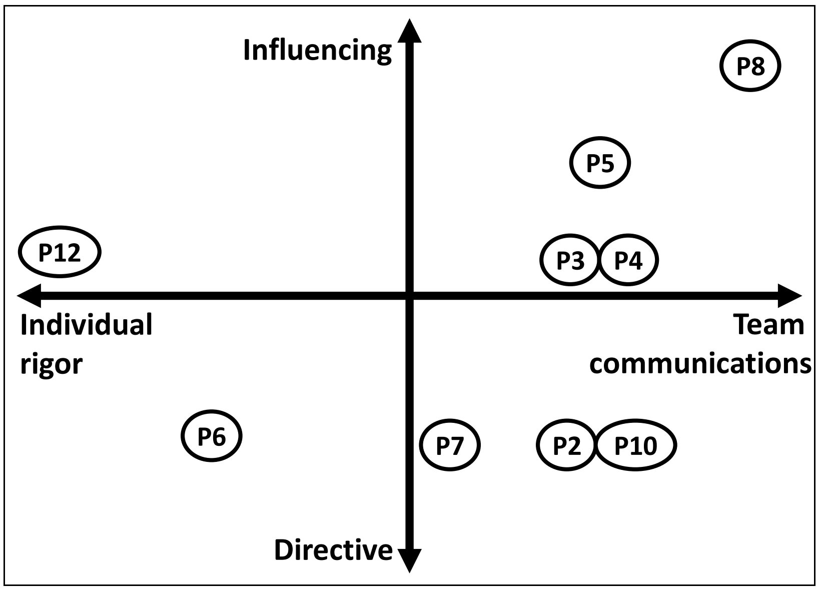


Figure 9: Expectation of team interaction

Though one might expect the choice of influencing vs directing to reflect the expert's authority in the organisation, in fact this was not necessarily the case. For example P5's role gave him authority and P7 was referring to peers.

5.6 Approach to Implementing Security

In terms of knowledge transfer and implementing app security the GT analysis showed two key categories: 'tick-box security', which evoked both positive and negative reactions; and 'whole system security', which was implied by many of the experts; there was also a surprising distinction on which counterparties the experts considered: potential attackers or project stakeholders such as product managers. These were more nuanced, reflecting differences in emphasis.

5.6.1 Checklists or Whole System Security?

Some experts preferred a checklist, excellent-coding attitude to security:

> *"Checklists I think are wonderful things. And if they are Why, How, What, Where, When, not just 'does it' – it's not just a 'yes / no'. It's a checklist that goes, in what way have you done this?"* (P5)

The experts expressed concern even about programmers who do appreciate the need for security. This is most typical of programmers working in projects with regulatory implications, such as the ‘Eurocard, MasterCard, Visa’ (EMV) rules for privacy. Most developers in that situation see their obligation as ‘satisfying the requirements’, and ticking the boxes on a list of ‘things that could go wrong’, rather than using these as tools to implement a secure system for the sake of the users and stakeholders who will suffer in the event of a breach.

> *Perhaps that is why [learning nitty gritty details of coding faults] is popular – it is relatively easy to read about in half an hour and say 'I understand what a buffer overflow is – I'm not going to do that anymore' (P6)*

Many interviewees stressed the importance of various aspects of Whole System security:

> *“I would just wish that education was better and that developers understood about separation of code and data and Saltzer and Schroeder’s 8 principles of computer security, and understood the background more and focussed less on the top 10 vulnerabilities – what they happen to be this year.” (P6)*

5.6.2 Concentrate on Attacker or Stakeholder?

There was an interesting distinction as to whether the emphasis was more on potential attackers, or on stakeholders such as product managers. Some emphasised the importance of understanding and reacting to different kinds of attackers:

> *“You also try and understand why someone is coming to your service in the first place. And try to give them what they want up front, so they lose interest and go away.” (P9)*

Others emphasised the importance of negotiation with stakeholders on what security was put in the product:

> *[When I started] a project I’d go back and ask [my customer] ... ‘how secure do you want it to be?’ (P1)*

Figure 10 shows these differences in emphasis.

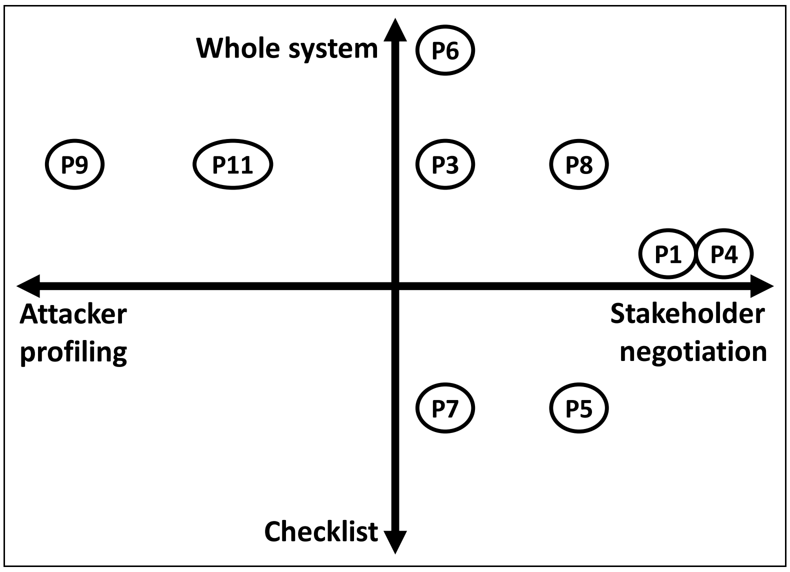


Figure 10: Preferred approach to app security

5.7 Summary and Implications

This chapter examined findings from our interviewees related to the research questions

> RQ2 *What motivated the experts themselves to learn software security; how did they do so; and how do they continue to learn?*

and

> *RQ4 How should we effectively introduce security to app development teams?*

Related to RQ2, we observed in section 5.3 a variety of different motivations for the experts to learn and continue learning software security. Related to RQ4, we observed in section 5.6 a lack of consistent emphasis on different secure app development techniques, and we observed in section 5.3 notable differences of opinion on how to motivate programmers to security, as highlighted by the spread of the points in Figure 8. Section 5.5 showed even stronger contrasts in experts' approaches to teamwork in Figure 9, and their approaches to app security in Figure 10.

The authors had experienced a similar lack of consistency in the early days of both the object oriented design paradigm (OOD) and the Agile development paradigm, each of which in due course converged into well accepted approaches: around UML and Scrum respectively. In the early days of each there were many good ideas and many experts championing different aspects of those ideas; the current situation in secure app development has a similar character. This suggests that the discipline of app development security is still at an early stage.

The convergence around UML and Scrum led to greatly increased programmer acceptance and knowledge of OOD and Agile development respectively. We suggest that similar convergence in app development security will lead to greatly improved programmer knowledge in that area. Looking at the history of object oriented design and agile development we believe two steps are likely to lead to this convergence. First is the codification of the main principles by well-respected experts in a popular form: a book, online resource or even video. Second is the championing of that codification by one or more large commercial organisations. Microsoft and Google are likely contenders, but both are tainted by their commitments to specific mobile platforms so it remains to be seen which organisation may champion a global approach.

6 Introduction to Dialectical Security

This chapter introduces our findings from the interviews on good strategies for programmers to produce secure app code. It discusses the method used, and theorises that each strategy represents a form of dialectical interaction. To explore these, it introduces three personas representing different types of programmer, and outlines the strategies themselves as a set of named techniques.

6.1 Introducing Dialectic

This chapter examines findings from our interviews related to the research question

RQ3 What are the most effective techniques to deliver app security?

Grounded Theory emphasises the creation of theory from data; the theory generated should cover the greatest variation in the data. Our initial GT analysis of the transcribed interviews suggested that our experts considered 'tick-box' security implied by standards like EMV to be insufficient and were proposing an approach to programming security that affects the full development lifecycle:

> *So implicit in [conventional thinking] is the notion that programmers decide what they are doing in code, which, to a degree, yes – it is how you might implement an algorithm – but the single biggest fault around that is, not*

> *around that question, but around programmers ... being told to put something in place without them understanding the greater implication. (P9)*

Our initial analysis therefore suggested that the experts were providing taxonomy of ‘whole system security’ techniques suitable for app developers. However on closer inspection we found that the interviews contained little mention of important parts of this taxonomy: for example devising mitigations or using checklists of possible errors.

Instead we observed that the core theme was the nature of a developer’s interaction with external parties. The word ‘dialectic’ had previously surfaced as a description for the review, penetration testing, and automated tool review aspects of good security practice [107]. ‘Dialectic’ means the finding out of knowledge, especially logical inconsistencies, through one person questioning another. The dialectical approach is best known as the technique used by the Greek philosopher Socrates in his dialogs; the reader may explore it in more detail from the extensive Wikipedia page on the subject [110]. It became clear that the friendly adversarial approach suggested by this term covers most of the other aspects stressed by our interviewees, and this led to us categorising the relevant techniques as ‘Dialectical Security’[3].

It was not hard to work out why dialectic is valuable. Programmers have to think what approaches an attacker might use to gain benefit from the system they are producing, and then to decide what to do to thwart those approaches.

> *Yes, the question is 'who is the attacker, who is the bad guy, who is the threat model you are dealing with?' (P3)*

This is very different from ‘normal programming’. Normal programming is about finding a good way to achieve a given set of functionality (as suggested by, say, Jackson et al. [57]). There is very little in normal programming about dealing with the attacks of unpleasant and possibly unwashed crooks.

> *They are very devious; there are exploits that they have realised which are, well, you wouldn't really think like that if you were an engineer (P2)*

[3] This has no connection with Marxist ‘Dialectical Materialism’, whose adversary is the unfolding of history.

The people who seem to be good at it (P9, P11) take a delight in the battle. Most programmers do not enjoy battles, being typically introverted and preferring cooperation [99]. Dialectical Security, however, provides techniques to take developers (whether developers, testers or other team members) and challenge them to make them think in ways that make for efficient security.

6.2 Documenting Good Practice

Since we are documenting good development practice, we use a format derived from the most effective way so far discovered to document programming and design practices: the 'pattern' format (see section 4.2). The author's own experience of using patterns for teaching and learning has led to modification to that format. He has given a number of seminars based on 'Small Memory' patterns [78]. Initially he used the conventional pattern approach, in which he described an abstract problem, a context and forces, and an abstract solution, and then moved on from there to specific 'known uses' and implementation notes. He found that this approach was difficult for programmers to relate to, and correspondingly saw little engagement. So he tried a different approach, which was to start with a specific example of a problem with a very specific solution; that is easy and concrete for programmers and technical teams to understand, and seems to be what they enjoy. Following that it was straightforward to widen the scope into something more like the pattern form: "when you think about it, you will see this is an example of a wider problem…" Then, grounded with the understanding of a particular use, he found the audience would follow the more abstract reasoning very happily.

We deduce that adopting a similar approach in writing will also lead to better understanding. Accordingly, to make the learning in this section easy to follow each description here starts with a concrete example of the full technique and moves out to explore wider implications. Each therefore starts with a particular illustration of both the problem and how it was solved, then generalises it to a more general problem. It then discusses a recommended solution to that problem, and ends with discussion of aspects of the technique. The context is the same for all these techniques: a development team or solo developer working to produce and support a software-based system.

6.3 Types of Programmer Discussed

To bring the discussion to life, this chapter uses three 'personas': composite characters who illustrate the issues being described in the interviews. The use of personas is well established in software interaction design, as discussed by Pruit and Grudin [86]. Microsoft has also used personas of software engineers [24]. Faily and Flechais [43] introduced their use in security research, creating the CAIRIS tool to create personas representing the security attitudes of software users.

The personas described here are not real people, nor composite representations of the interviewees. Rather, they are typical of people described in the interviews, the people with whom the interviewees work. The personas are Jane Solo, Rob Youngcorporate and Jo Socialnetwork.

6.3.1 Jane Solo

Jane Solo is an example of an isolated programmer. Two of our interviewees (P1, P12) were themselves isolated programmers – though unusually security-aware ones – and this persona is based on their narratives and those who have worked with people in similar positions (P5, P9). Such programmers tend to be highly motivated to improve their skills, as found by Enes [40], but unaware of the issues around security and privacy.

> *"It's not that [programmers] have passed judgement on [app security], and that it is unimportant – they just don't realise that it is important"* (P5)

Persona – Jane Solo

Jane Solo is an experienced app programmer. She works on a contract basis, working sometimes on her own doing one-person projects for customers, sometimes working with teams of other app developers – either remotely from home or in their offices. Her projects tend to be technologically quite interesting: combining beacons with social media, for example, or creating NFC hand scanners for baggage operators. When she does not have contract work, she works on improvements to a game app she has produced and sells via the App Stores.

Jane does not think much about app security in her projects. Her clients do not worry much about privacy or app misuse; they tend to be inexperienced in the app world and rely on experts like Jane to advise them. Jane reads about issues like the US Office of Personnel Management loss of personal details and the Edward Snowden affair, but knows they are the result of security leaks from large servers; she is not writing any server code, so they are not close enough to home for her to be concerned about them. She is, though, highly motivated to improve her skill; her career requires her to find new work often, and every relevant skill she can develop increases her saleability to new employers.

Jane is unaware that her lack of interest in security could come with any costs. She does not know that her reuse of credentials in her game allows hackers to see all the players, along with sensitive information related to them. She does not know that the app she is producing for her employer collects credit card information in a way that can be seen by malware on the phone. Indeed it is unlikely either issue will cause her or her employer any embarrassment any time soon: even if credit card details are stolen it will take the card networks a long time to work out it was her app that allowed it. However Jane does care very much about her customers and about the people for whom she produces the apps – she is proud to be doing the best for them, and would not want to cause them any harm.

6.3.2 Rob Youngcorporate

Our second persona, Rob Youngcorporate, represents programmers working for companies with a significant software development capacity. Such companies may be aware of security issues around their commercial software generally, but will generally be unaware of the specific implications for app development.

P6, P8, P9 and to some extent P10 were all discussing programmers in this kind of organisation. While the interviewees themselves have a strong understanding of security issues, many of the programmers they will have worked with and continue to work with are less well experienced, and may not have the resources and infrastructure to find out about security issues.

Persona – Rob Youngcorporate

Robert (‘Rob’) Youngcorporate works for a company that provides a traditional service: insurance. The company has looked at what the competition are doing, and concluded they need an app. Robert is two years out of university and confident in his abilities and he's happily taken on the learning task and the job of producing such an app.

To get started, Rob produced a list of topics he would have to learn up to get started on the app. Naturally learning about mobile development languages and development environments came top of the list, and he's read several books and been on a training course on the subject. He has also learned a great deal from his colleagues about the company's systems and web APIs to which he will be integrating. However he also realised that for an app dealing in money there would be security implications. Accordingly he read the mobile OS manufacturer websites, and looked into the EMV standards for apps. This second line of investigation did not get him very far; EMV had not produced standards for apps at that time. So he read the OWASP web pages on the 10 major app issues [120] instead .

Rob believes he has 'covered' security for his app development. He does not know that he's using one of the payment plugins in a way that causes it to log sensitive information to a file on the phone, nor that the PIN code he's implemented for the login process would be open to a fairly simple brute force attack, allowing a phone thief to learn card payment details from the app. Rob cares about his company's reputation, and certainly would not want such problems to become public knowledge; he would also be concerned about the effect on the individual users.

6.3.3 Jo Socialnetwork

Our third persona represents the minority of programmers working at organisations that are well aware of security and privacy implications and consider if part of their commercial offering. Both P3 and P11 work for companies similar to that described; P2 and P7 work for other kinds of companies that are also very experienced in software security; and P5 built up a high level of security expertise in his organisation. The discussion of these interviewees was mainly about the best practice approaches they and their companies have developed.

> *Yes, so [my company] has a strong culture of code review, nothing gets submitted without it being reviewed by at least another engineer. And there are strong processes to protect that fact. And there have been a number of*

times when either in code review or design review, designs have come in, and I have been able to go 'Hang on a minute, look at this, or do that'. ... And so catching those things in code or design review before they go out, so we don't reveal [damaging personal information] any more. It's the sort of thing that a security mentality helps with. (P3)

Persona – Jo Socialnetwork

Jo Socialnetwork works at a large social-networking company. The company has been around ten years or so and has millions of users, many of them paying. They have seen their competition stung by security and privacy breaches, and had one or two of them themselves. As a result, the company takes both software security and software privacy very seriously indeed. The company has a strong corporate culture, and all the programmers know ‘the way we do things here’: through discussions with colleagues, through documents, and through reviews and project retrospectives. Jo has been at the company over a year, and knows exactly what is expected by way of privacy and security.

Every one of Jo’s projects starts with a review of possible security and privacy issues and a discussion of possible exploits that might be undertaken against it. Throughout the development Jo knows that she can at any time request a security review or express concerns about security issues, and these will be taken seriously and acted upon. She and her team are encouraged to think up potential problems, and rewarded for finding security and privacy defects. Before Jo's code is released, it will be reviewed by someone with experience of software security, to reduce the possibility of errors. However, these rarely happen at release time, because Jo, like her colleagues, has a good understanding of security issues, and she regularly calls upon colleagues if she is in any doubt.

6.4 The Techniques of Dialectical Security

The following sections describe six techniques of Dialectical Security. The techniques are:

Brainstorming the Enemy	Ideation sessions working with stakeholders, penetration testing experts and others to derive possible attackers and attacks on the system
Commercial negotiation	Communicating security decisions in ways their stakeholders can understand, to prioritise them against other requirements
Cross-team security discussion	Effective communication with other development teams to ensure security
Security challenge	Using professional and in-team security experts for code reviews and penetration testing
Automated challenge	Using automated tools to query possible security weaknesses
Responsive development	Gathering continuous feedback from the use of the system, and responding with continuous upgrades and interactive defenses.

In mapping these techniques, we found that unlike many collections of software security patterns, such as Schumacher et al. [94], these do not break down as steps to be carried out as part of a process, nor do they form a hierarchy. Instead they characterise themselves in terms of the source of the challenge to the programming team: other team members; tools; other roles in the software development process; and end users and the consequences of end use. In each case, the dialectic can continue throughout the development cycle, and in each case it is always two-way: the increase in security comes from the interaction with the challenger, not from a passive understanding of the challenge.

Figure 11 shows how the techniques relate to different counterparties to the development team. Ovals are the techniques; arrows show the most important interaction for each. The illustration only identifies the key counterparties for each interaction; the other roles are involved in most of them: for example Trading Security Requirements would normally include project management and software architects as well. Note that in Brainstorming the Enemy the major source of the 'dialectic' challenge is from other team members; this is shown with a double arrow.

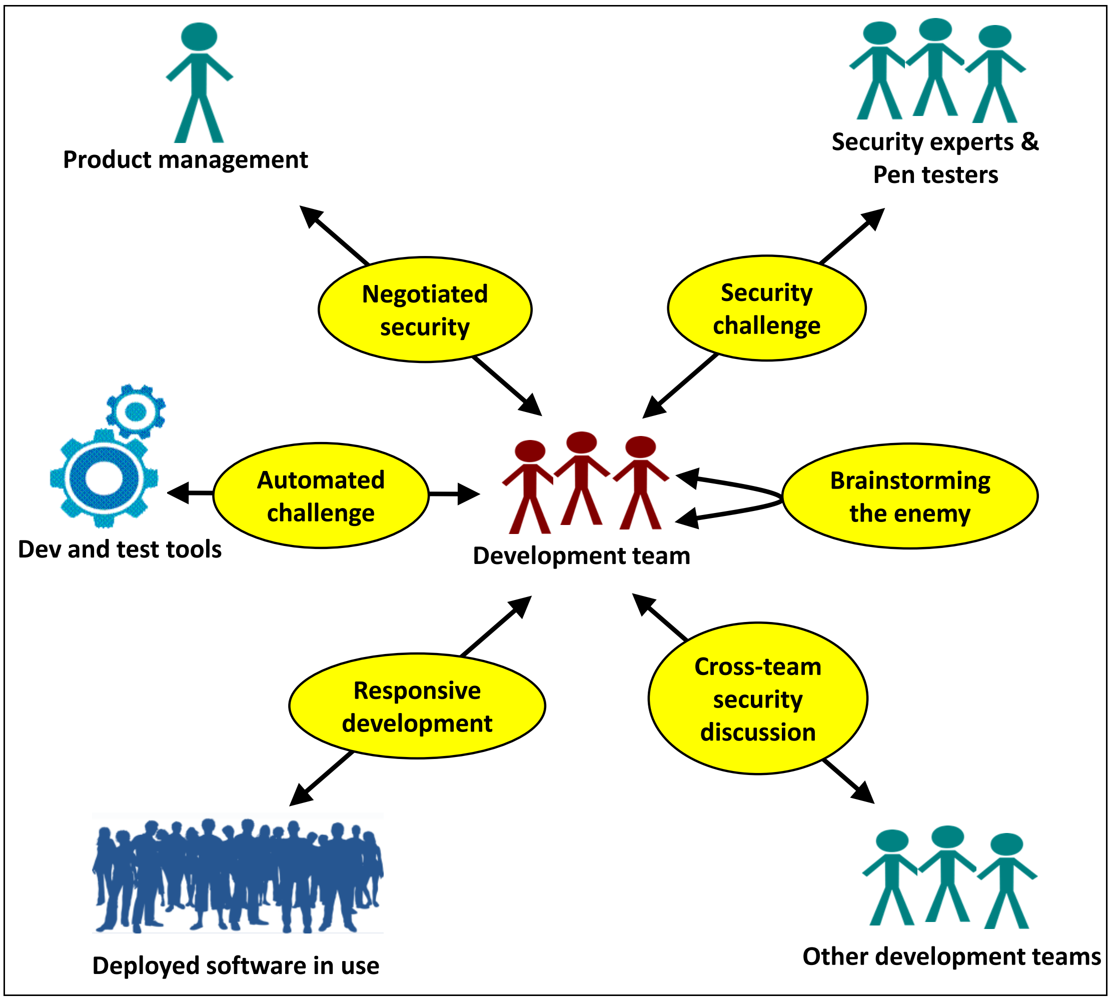


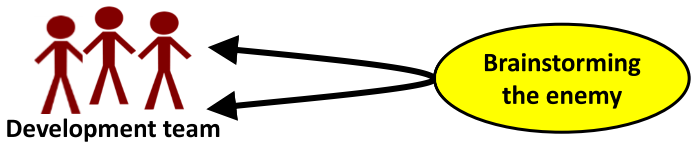
Figure 11: Techniques as dialectic interactions

Note that we do not have evidence to claim these techniques are the *best* techniques for achieving security; however the statements of our interviewees certainly provide a strong indication that these are *good* and *effective* techniques.

7 Techniques of Dialectical Security

This chapter explores each of the six techniques of Dialectical Security as sections 7.1 through 7.6, discussing when each is suitable and how it is carried out, and illustrating each with examples using the developer personas.

7.1 Technique 1: Brainstorming the Enemy



"One of the things I like to do with the [penetration testing] guys is to, if you sit down and say 'what are all the different ways you could subvert this system'. It is quite common to come up with 20, 30, 40, 50 in five or ten minutes of brainstorming. I bet you, you wouldn't think of half of them." (P2)

7.1.1 Example

Jo Socialnetwork recently started a new project with her team, implementing an enhancement to the payments collection functionality to support payments via PayPal. As she started, she realised she has a problem: how can the team implement security if they do not know against what they are protecting? So as they started the project, one of the first things they set up was a half-day session with the team, penetration testers, the product manager and a representative from PayPal to understand what the attackers might be, their motivations, what they might be after and how they might typically get it. Of course, they had the lists of similar considerations from earlier implementations of the payment collection functionality and they used them for reference. Based on these lists they brainstormed a new list of possible attacks on the new system.

7.1.2 Exploration

Any system can be broken with sufficient determination, ingenuity and resources.

> *Every security system can be broken. Period. There are even ways of getting the certificates off a phone, by freezing the phone and reading the memory. There is nothing you can do to stop a truly determined person to getting in, short of dropping it into a nuclear furnace. The best you can do is make it difficult enough for them, that they will lose interest – that it's not worth the trouble. (P7)*

> *I quickly realised that no system is ever unbreakable (P9)*

Secure app development is therefore not a matter of making a completely secure system. Instead it becomes a question of which defences to put in; where one should spend the time and effort defending the system to deter the largest and most damaging potential exploits. Making those choices requires an understanding of the potential attackers:

> *I think it is actually very important to understand the motivations behind why somebody is hacking the system. We try to address the motivations of the attackers, versus the technical aspects - just locking it down for the sake of locking it down. (P11)*

It also requires an understanding of what attacks they might make:

> *I think the things that are the most challenging around security really are trying to understand the threat landscape and trying to understand how threats are realised. (P2).*

Thinking about: where could this go wrong? That is the thing the people just coming out of university don't understand, don't think about is what are the attack points of this particular code, what are the failure points, even more than attack points, because anything is an attack point. (P7)

Neither attacker profiles not attack descriptions, however, are conventional knowledge for an app developer. So how do they best obtain them?

7.1.3 Solution

Identify both attackers and possible exploits in two steps. The first step is to create profiles of likely attackers. This means querying experience with similar products, discussing with others in the industry, and consulting experts. The attackers may not be the obvious ones:

There are clear reasons why someone would want to attack a bank, but actually the real reasons for attacking a bank are very seldom to do with trying to get financial rewards. It is much more around what information you can get about people. Banks hold information about people. So [it might be] a private investigator who is trying to track someone, or a hostage situation, where people might have done things, or simply learning more about behaviour. (P9)

The second step is to use brainstorming sessions for attack profiling.

I was involved in a lot of conversations about trying to think about doing really evil things, so I think in order to protect people from harm we have to think about how harm can be done. So, brain-storming bad intent is part of the life, really. (P5)

These brainstorming sessions include people with different roles, especially testers, penetration testers, app security code reviewers and security specialists. An excellent concise recipe for running them is in the seminal work on negotiation, Fisher et al.'s 'Getting to Yes' [44], whose chapter 'Invent Options for Mutual Gain' contains a step-by-step prescription for an effective brainstorming process.

Particularly with development teams using agile approaches, this ideation process continues informally throughout the initial development project, and into the subsequent deployment and later lifetime of the product. The most security-capable teams included attacks and motivations found in the course of deploying the app, or afterwards.

> *The other thing, is ... [to] reward proactive thinking and this is two levels of that: trying to think what could happen next, how could it go wrong, what am I missing, but then the next level of reward, is rewarding people for research. And thinking about how to do harm. Actively encourage them to think like a hacker. (P5)*

7.1.4 Discussion

Even an apparently innocuous product may be under threat:

> *The only question is 'are you, as a target, worth it?' So, that then becomes very much the entire world's permutations of life, because you might be selling flowers, why would you be a hacking target. Well, if it is a spotty faced teenager wanting to know whether his girlfriend has had flowers delivered that he didn't send, then you have information, and now there is a reason. (P9)*

There are tools available which may help

> *Microsoft have a threat modelling tool, which they make freely available, and it is actually a really nifty thing that you can draw a data flow diagram of your system and it provides a framework where you can think about what are the security implications of this bit of flow and this interface and what are the other security domains. (P6)*

Of course the degree of formality and effort involved depends entirely on the context. A solo app developer considering the privacy implications for a new game need not probably worry about a very formal approach; a development team producing a social network extension for a banking application will need a formal documented record of the threat motivations, personas involved, identified potential exploits. Moreover, the latter will need to be correlated in due course with the proposed risk analysis and mitigations and extended throughout the lifetime of the product.

There has been some work studying the use of brainstorming sessions in software. Shih et al. [95] looked at the use of brainstorming at Microsoft, and highlight some of the problems to avoid. Dashti and Basin's paper on Security Testing [32] describes a process of finding possible security exploits, which reads similarly to this Brainstormed Profiles technique.

7.2 Technique 2: Negotiated Security

Development team
Negotiated security
Product management

For businesses it is a risk based approach which they need to understand and neither [management nor programmers] should be caring about actual nitty gritty details of coding which is just an artefact of the whole thing. (P6)

7.2.1 Example

Once Jo Socialnetwork had the list of issues from her first Brainstorming the Enemy workshop, she found the team could have enough work implementing mitigations to keep them working well past the planned product delivery date.

That clearly was not acceptable, so she and the team then used the profiles of the attackers and her team's experience to estimate how likely each exploit might be, and the possible impact to the company of each. This required discussion with the product manager, security and risk specialists and occasionally more senior management; and as a result they decided to ignore some of the identified issues altogether. For the remainder, the development team then had sessions thinking up a variety of possible 'mitigations' to implement for each risk, and estimating the effectiveness and development cost for each.

Jo then discussed the list of attacks, impacts and possible mitigations with the product manager. Based on that, the product manager and senior management then made calculated risk-based decisions comparing the 'value' of each mitigation to their business against the value of other enhancements: functionality, performance and the like. Since they use an agile development process, the implementation of the 'mitigations', where these were in code, was included in the product backlog to be implemented in due priority order relative to work. As the project continued, and the team identified further possible attacks and mitigations based on feedback from testers, users and others, these too were prioritised based the associated risks.

7.2.2 Exploration

Merely identifying the possible attackers and exploits does not of itself deliver app software security. The need is to prevent them causing significant damage to users, stakeholders or others. To achieve that, a development team takes the list of possible attacks, and work out possible mitigations for each. These mitigations will each have costs in development time, commitment, finance and sometimes usability. The team can estimate financial and other costs for each. However the decision of what aspects of security to implement is a commercial one. Implied in every decision about software security is a trade-off of the cost of the security against the benefit received. Every security enhancement needs to be weighed against other uses of the investment (financial, time, usability) required. For example,

> *[Costly development approaches aren't] suitable for a lot of start-ups. And the same goes for security. You're going to have to make a security decision upfront. (P1)*

How, then, do the developers make the decision which security enhancements to implement?

7.2.3 Solution

Interpret the security risks and costs to stakeholders (project managers, senior management, customers) in terms they can understand and use to prioritise security concerns against other organisation and project needs.

> *[When I started] a project I'd go back and ask [the customer]…'You do realise this [information] can be seen'. It goes from there: 'how secure do you want it to be?' You have to show that there's a problem first I think" (P1)*

It is hard to over-emphasise the value of such interpretation. Many of our interviewees made the point that 'security is not an absolute' – security is what the users and stakeholders need for a particular situation at a particular time. For such stakeholders to make a good decision on what they require requires particularly effective communication. The stakeholders will be making cost benefit trade-offs comparing various business risks.

> *You've got to put a weighting on the threat. You've got a level of threat, and you've got to put the appropriate level of security against that. (P4)*

There are techniques available to give objective assessment of security risks, such as work by ben Othmane et al. [80]. Vitally – and several interviewees stressed this – the cost-benefit trade-offs mean that perfect security, even if possible, would rarely be a good business decision:

> *And actually the way this works, in practice is you have to do less than a perfect job, in order to have a measureable degree of failure or fraud or whatever, so that you can adjust your investment and say 'I am managing this to an economically viable level' because if it is zero, you have invested too much. (P6)*

For simpler projects and systems, there may not be sufficient engagement from stakeholders to be able to do this kind of trade-off; in that case it becomes the responsibility of the developer:

> *[Often it's impossible to get signoff on security in a big company and so the decision is usually down the developer because you can't get the signoff. And in a small company may just be the same]. Customers often don't have a view. The important thing is making the decision. (P1)*

Given that each mitigation now has a cost and benefit, the decision on whether to do it becomes part of standard project management process. It is outside the scope of our theory – and indeed of the topic of software security – to explore how these decisions are made; the balancing of risk cost and reward is a well understood aspect of business life.

> *And it has to be a bit of a trade off as well in terms of business. You've got to make the trade off as to what's good for getting a solution available now, and having one available in a year's time, which no one will buy, because everyone's gone with one which doesn't even consider security at all. (P12)*

> *There has to be a system level thinking going on about where you do certain things and might not do certain things but you ultimately have to think [like a risk manager] (P5)*

7.2.4 Discussion

Interpreting a threat to a non-programmer involves putting it in terms that are meaningful for them:

> *I think you normally phrase it along; you do realise this [information] can be seen. It goes from there: how secure do you want it to be. You have to show that there's a problem first I think, that's how it's phrased. This particular bit of data at the moment, at the way it's going to be implemented, a certain kind of person would be able to see that. Are you happy with that? From there you go to what other data is there. (P1)*

It is important to use language that is appropriate to the problem; phrases like "*it's insecure*" can actually hinder, not help, communication:

> *Half the time you're going to be a bit careful about what you say because obviously if you say something like 'the keys are insecure' – meaning insecure from the point of view of some FIPS attacker – then you are going to freak out somebody who is just trying to install a lock on a bike shed. So I think you have to get the right perspective there. The threat level and the information level, and again, I don't really want to hide anything, the number of things I've seen go pear shaped because somebody has said something stupid like 'it's insecure'. (P4)*

Estimating the probability of a threat can be surprisingly straightforward, in the experience of the author [108]. One estimates as 'low', 'medium', or 'high' for both the probability of each exploit, and also for its likely impact. Though crude, this gives sufficient information to reason about the attacks. For example, if one is creating a game app, it is unlikely that the hacking departments of major nation states are going to be especially interested (unless you're building Angry Birds, perhaps). So the probability of sophisticated attacks is low.

Within simpler projects, the act of thinking through mitigations may itself be valuable even without negotiation:

> *I certainly wouldn't go as far as saying that everything I've done is absolutely [or even] sufficiently secure… It's that for the most part I've got*

an idea of what the risks are so if there is something which starts to need mitigating, I can think about mitigating rather than [it being an open problem]. (P12)

The discussion of the value of implementing a mitigation needs to take into account that some are more effective than others are. Mitigations handling general problems are more valuable than fixes for specific issues; P11 provided an interesting overview of how his team approaches the problem:

And basically our team has four primary goals [for mitigations]: exploit mediation: we try to make it so that even if there is a bug that you can't do anything bad with the bug; exploit containments, so we recognise that people are going to get through our mitigations so let's try to make sure that that is appropriately contained; attack surface reductions, so from a security point of view, if you can't reach the code, well there could be a bug but it is useless; and probably more relevant here to application developers, safe by default settings so we make it that you have to go out of your way to introduce a security hole. (P11)

P11 also pointed out that the solution to a security weakness need not be in software:

And the analogy I use all the time is: if you look at the world, you and I are both vulnerable to Ebola; we are not immune to it. If you were exposed to Ebola, you would get infected by it. So you have a security vulnerability; you have a medical vulnerability; I have a medical vulnerability. Yet the medical industry and the newspaper industry and the press don't go about publishing articles that say 100% of population is vulnerable to Ebola. And why? Because the medical industry has concepts of quarantine and are able to control populations, control the propagation of diseases. They have defence in depth; they have for certain diseases like smallpox this concept of an inoculation, a shot. (P11)

Indeed many organisations approach the subject of risk (of which security risk is one aspect) as a discipline in its own right:

Have you ever worked with a risk manager, a really good corporate risk manager? ... These guys are part mathematician, part bookie, part process/problem solver (P5).

One approach to expressing risk in a way that is useful and meaningful to stakeholders is to express it in terms of whether the risk is increasing or decreasing – an approach used by William Brandon, currently CISO of the Bank of England [121]. One can speculate that this works because stakeholders usually have a view of the current risk levels as acceptable or unacceptable, and can therefore reason about the results of them improving or worsening.

It is important to remember that this is about the calculated estimate of risk. The fact that an unlikely bad event subsequently happens would not invalidate a decision not to mitigate against it. The original decision would only be 'bad' if it was based on sloppy information gathering or faulty thinking.

[Following the decision to ignore a risk], the important thing was that you pointed out the risk first, so when it happened you just go 'yes, it was a commercial decision, move on'. It is in your risk log, and it is agreed upon – it is understood. (P8)

One aspect of Negotiated Security is that it may turn out to be appropriate sometimes *not* to pay for security; unsurprisingly the total need for security may turn out to be less than a purist security expert would expect. Equally it is important to present the security budget as a positive aspect:

There is a budget for security that every company must have – that budget has to include resources that a hard core manager would say are stolen from you, a more enlightened manager would say 'it's the tax on not being attacked'. Giving those resources to your attackers and playing the game, you have got to keep them amused, you can't just give them to them; they want a reward of some sort, they are after something, so you have to play the game with them. But that is the cost, that is your budget, and there is an element of network capacity, there is an element of disc space, there is an element of servers, there is an element of security specialists, there are the *managers and the monitors. All of that around, that is part of your budget for security, if you don't have that, you are going to fall foul of hacking. (P9)*

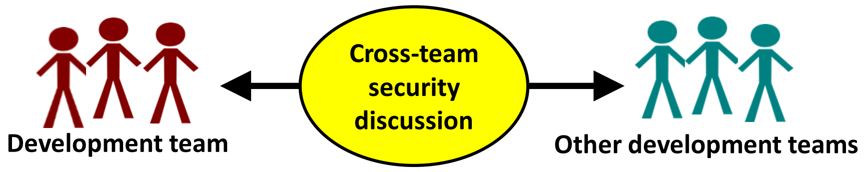
Finally there are some perverse incentives against security, highlighted by P6, which begin to give some idea why software security is often seen as less important than one might expect:

George Akerlof is a Nobel Prize winning economist, and he wrote a paper in 1970 called The Market for Lemons [4]*, and what he was talking about was used cars in America. So, you buy a used car and it could be a peach – little old lady drives it to church on Sundays, or it could be a lemon, which is going to go wrong next week, and the consumer cannot tell the difference. So the economic consequences of that, is that, as a vendor, you keep the peaches, because you can tell the difference, …and you sell the lemons, and over time, the consumer comes to expect more and more lemons, so they are willing to spend less and less, so it ends up being a race to the bottom. So if there is some product attribute that the consumer cannot measure, you tend to get less of it over time because there is no economic incentive to keep putting it in. And security, sadly, has a lot of that aspect. If I buy a product off the shelf, I can't say "that one's got security and that one hasn't" So I'm just going to pay the value of the one that hasn't. And therefore there is no economic incentive for the manufacture of that product to build in any extra security, because it's not going to get them any more sales. (P6)*

Moreover the straightforward commercial incentives are not always as clear cut as some security proponents might suggest:

I looked and again there is anecdotal received wisdom that lots of companies have gone out of business because of security breaches, and it is just not true. I found two, only two, ever, that I could find that had gone out of business due to security breaches and one of them was DigiNotar who were in the security business. They were a certificate authority, so they clearly did go out of business when their fundamental offering was broken. And the other one was … an ISP in the USA, and … they were attacked and their business was entirely based around Amazon web services and the attackers compromised *their keys – basically wiped everything they had on Amazon web services, and they had no other back up of their system – so it completely destroyed the company. It was bad computer security yes, but the fundamental thing that put them out of business was that they didn't have any backups. (P6)*

7.3 Technique 3: Cross-Team Security Discussion



And I think, ... [what was very successful was] around even [specific security issues and mitigations] – the working incredibly closely as a team, and just having very open discussions with cards on the table and removing the fear around discussing aspects of security which, I often find in project meetings, people don't want to bring up because they feel they don't want to expose their own domain. (P8)

7.3.1 Example

Rob Youngcorporate had identified several security issues he knew his product would need to handle. He knew he would need an authentication process for each user starting using the app, and some form of 'quick authentication' thereafter. He knew there would be a risk of 'Man in the Middle' (MITM) attacks on the app's communication with his insurance company's back end services.

So he discussed the problems and the potential security attacks he had identified with the User Experience team, and worked with them to identify a suitable way to give the user this 'quick authentication' by comparison with similar apps (banking, for example). He also talked a good deal with the implementers of the back end services, agreeing HTTPS-based secure protocols and helping them prevent 'brute force' attacks such as trying thousands of possible passwords for a given user.

7.3.2 Exploration

Many security issues span a number of teams: development teams, operations and even marketing or publicity. Thus there is a frequent danger that security problems can 'fall between two stools', remaining ignored because two teams each think the other is responsible for the problem.

The problem is exacerbated if the development team are not natural communicators:

> *I had a core technology group ... who worked for me, and these guys were double firsts in maths from Cambridge. Incredibly bright guys: appalling interpersonal skills. (P5)*

And sometimes by organisational politics:

> *You get teams of people who are perhaps very protective of their platforms, because they own the system and they are master of the system, and they want it to be seen as a golden system… Quite often the people representing the system are perhaps one step removed from the real hands-on techies – they are generally a manager, who ultimately becomes associated with this platform and they feel that their role can be at risk if that platform was ever to be undermined and another platform selected over it, so they wouldn't, by default, become the owner of that system, so the silos become self-reinforcing but it is very difficult sometimes to know whether you have actually been delivered all the facts. (P8)*

And also if teams are effectively separated by time – they're not working on the project at the same time:

> *[There is a big] difference between the operational and project approaches. [And security is the one real thing that is not going to get handled by that handover]. That is a real challenge. (P8)*

7.3.3 Solution

Ensure frequent and open communication on security problems in any way available. Bringing members of the different teams together on a social basis encourages that kind of communication:

I am a strong believer in the social aspect of it… I think if you can bring people together physically on a regular basis so that you can get to the stage where people are discussing family, friends with each other and everything else, it breaks down a lot of the artificial barriers that are there. … I do think co-location was key, and we would regularly come together, we would share a whiteboard and we all had the same view of the world. Openness and transparency - I think it makes a huge difference. I really do. (P8)

So does encouraging informal communication on technical issues:

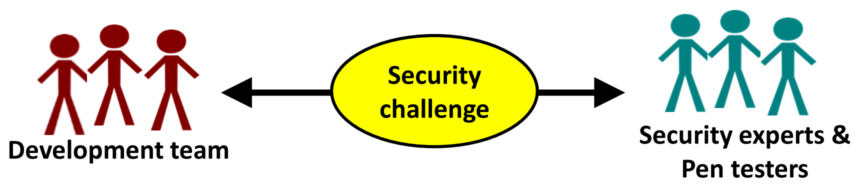
[Of a successful project] I guess we were working with a team who were experienced but also everybody who was close to the project, lived through the project life cycle to delivery, were very comfortable picking the phone up to anybody else and discussing any aspect, and everyone reported back quite openly what they were seeing when we came together. (P8)

An effective but very different form of communication is the more formal documentation of responsibilities. One straightforward way to do this is a ‘Security scope’ document that identifies the security responsibilities of a given team. That highlights where ‘falls between two stools’ problems may happen, and is used, for example, in a secure development process introduced by the author [108].

Where multiple organisations are involved, this may even be contractual:

We have got in our contract with [our development company] a definitive list of things that they will have failed to do their job if they haven't protected against these types of attacks. When we find a new one, we try to write a test for it, we put it into the document (P5)

7.4 Technique 4: Security Challenge



"Nothing gets submitted without it being reviewed by at least another engineer. And there are strong processes to protect that fact. ... The most successful technique has to be review by [a security] expert – you can't really beat that – an actual conversational review by an expert, because someone who is an expert in security might not be an expert in the domain." (P3)

7.4.1 Example

Jo Socialnetwork knows how to ensure good security with her code. It is built into the very processes and mentality she and her colleagues use in their development.

First, any time that she or any of her colleagues may have a concern about a security or privacy issue, she knows anyone can flag it to the project manager who must almost immediately set up a review with the appropriate people from the security team.

Secondly a lot of her work is pair programming, and she and her 'pair' continually ask each other questions, including questions about security; just as they help and remind each other to handle all of the aspects of code – especially security and privacy.

Finally there is a lot at stake for the company, so she knows that, as part of the release process, all of the changes she produces will be reviewed by a separate security team before they are made live.

7.4.2 Exploration

It is notoriously difficult to spot one's own errors – c.f. Meyers' Principle 2 of Software Testing [73]. This is especially true when the errors are faults in complex reasoning, or are due to misunderstandings. A programmer working solo is likely to create avoidable security problems, just because they can naturally have only one point of view.

So it is very easy when you are trying to deliver something yourself, as a developer, to pass over the bit that you are not doing (P5)

This problem extends to programming teams; a team, too, will always to some extent suffer from 'groupthink'; the need to generate a shared understanding brings with it the danger that that understanding may include misunderstandings and blind spots.

7.4.3 Solution

Set up the development so that each has another person or team with a different viewpoint challenging the security and privacy aspect of assumptions, decisions and code.

There are several common ways of arranging this within a typical development process: pair programming, security review, code review and penetration testing.

Although not normally cited as a security technique, Pair Programming gives the developer the benefit of external questioning; it also enables a programmer to handle more complexity during the programming process; two people can keep track of more issues than one.

> *Two heads are better than one, more eyes on the problem. (P7)*

A security review of the design, technologies and protocols of a system, by an experienced secure software expert, is particularly effective, and also helps developers to learn more of their code base [90].

> *There is a separate Security Review system, so if you are doing code that impacts security in your judgement, it goes to people who are security experts who will do the security review and they find stuff. ... And anyone involved can say, ' this needs a security review' – it might be the product manager, who is representing the user, it might be one of the software engineers who is writing the stuff, it might be one of the code reviewers who is reviewing the software. (P3)*

Sophisticated organisations may even separate the security and privacy concerns completely:

> *There are also Privacy Review Processes …. And again you go to Privacy Review, and they say 'representing the user and their control of data, should we release this?' (P3)*

For a cloud-based system, the widely accepted way of ensuring security is Penetration Testing, where an external 'white hat' security team simulates what an attacker would do to attempt to gain access or disable the service. They then feed any 'successful' exploits they have found back to the development and operations teams.

[Ensuring that the teams that I'm working with produce secure software] tends to get handed off, in most companies I've worked with, to a white-hat hacking team. [They] don't do it at a code level. (P7)

At the operating system level, one can also penetration test a mobile device:

"I think the one [approach] that has been, arguably, most useful has been using specialist external consultancy around security. Not for training, but 'can you just come in and penetration test this device'" (P2)

The widely used equivalent of penetration testing for an app is an external security code review. Many companies now specialise in this kind of app security code review; they gather lists of known security issues found in apps, with mitigations for each, and then review the provided code to look for those security issues. Security code reviews are also very effective when internal to a company:

Code review is what we do endlessly. We certainly do not let any form of code out the door, without an independent review and that is eyeballs on the code and that is discussion about the code (P5)

We do code reviews as much as possible. And I point out when I think something may have some issues, things like that. (P7)

All of these approaches are expensive; there is a significant resource cost to providing the challenge. In the case of pair programming, research suggests that the net cost is relatively small [25]. The other three interventions all represent additional costs for an organisation, however, which need to be traded against the corresponding benefits:

You call them out, but ultimately [best] is code level reviews but again it is this balance between the ideal world and the timescale, versus the risk and the consequences of the risk, or the consequences of an attack (P7)

Because of the need for the involvement of other people Security Challenge does not usually make sense for solo programmers, or those working in organisations that do not take security seriously.

7.4.4 Discussion

Penetration testing in particular has significant limitations:

> *I think what it struggles to address is the interpretation that comes about between a design and an implementation on potentially on the back end side of things. When you have got live systems that perhaps is already doing something broadly similar – maybe they will have the flow of request reversing – one system requests an address, then a phone number. Another may already have a piece of code written that requests a phone number and then an address. And it's a spurious example, but it shows how quite easily you may have specified these things in this order for a very good reason. And as far as you are aware, they are copying a design you have created, but actually somebody there sees an opportunity there to re-use something they have had sitting there for 5 years, and so calls that system [and thereby generated a security problem that Penetration Testing wouldn't find]. (P7)*

Therefore it's not always very effective commercially:

> *You can do your penetration testing or your external testing as much as you like but actually it doesn't really tell you the likelihood of the next breach. (P6)*

P6 even suggested that Penetration Testing is likely to go out of fashion as the main approach:

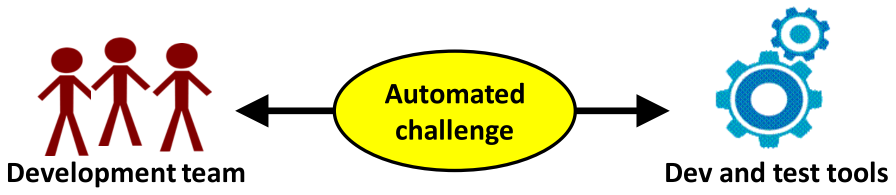
> *There are waves of approach to defending the security of computer systems, in fact anti-virus was the 1990's wave. Today's wave is definitely penetration testing and code inspection and all this kind of focusing on vulnerabilities and I think that's a wave and it will have to be superseded by something else and in my view that something else has got to be about, essentially, about development processes. (P6)*

P5 has an interesting approach to code security reviews, where two reviewers work together relatively informally:

> *Every now and then, Joe and I will be looking at something in somebody's code, and I can see – and Joe loves his boys – and I love them as well, and girls, but his body language – he should never play poker with anybody – he'll be going through the code and he just pauses on the down button, just long enough – and I'll go 'what are you seeing?' And he just looks at me and goes 'Well, I was just thinking about that'. (P5)*

There is a good deal of literature about code reviews generally. Recent studies include Baum et al.'s analysis of code review in industry [13], which stresses the importance of making reviews a part of the normal software development process. Rigby and Bird's study [90] recommends in addition having two reviewers, making the reviews constructive with an emphasis on fixing problems, and doing the reviews as part of the release process. The OWASP Code Review Guide, a book [81], contains detailed discussion and recommendations how to carry out a security review.

7.5 Technique 5: Automated Challenge



> *"[The most successful technique I have found is] to use various types of Lint checkers" (P7)*

7.5.1 Example

Jane Solo does not worry much about app security, but she does care a good deal about the professionalism of her approach. One of the things she has discovered is that the error messages she gets from the normal compilation process are not very helpful in tracking down defects. She usually includes some additional checking tools in her development process to point out further defects that may be present.

As these checking tools are improving, she notices that some of the defects they are highlighting are in fact security defects. These lead her to wonder if there are other aspects of security that she might need to consider.

7.5.2 Exploration

Security Challenge can be very effective, but it is costly in human effort and impractical in many situations. Few solo app developers, for example, will have the money to pay for an external review of their code, or the social capital to persuade colleagues to do so. Likewise many organisations do not see value in paying for penetration testing or external reviewers, nor have skills to do either in-house.

Equally, it is poor use of expensive resources to find problems that can be cheaply found elsewhere.

How do we achieve this?

7.5.3 Solution

Use software tools to create dialectical challenges to the programmers. There are two areas where automation can help a great deal with the development of secure software. These are automated code analysis, and automated security testing.

Automated code analysis acts as an extension to the compilation process of the code, and looks for possible security flaws in the written code. Tools to do this are sometimes called ‘lint’ checkers, after a UNIX tool that does extra checking for C code. There are now many such tools, some produced by commercial companies, supporting different languages and purposes:

> *We use something called Sonar* [97] *which is a code inspection tool we'd written templates and guides for our coding standards and certain patterns we are looking for in a code and we are looking for changes in the code that are greater than a certain percentage and there are specific bits of the code we are looking for any change that should never happen. (P5)*

They are excellent for looking for common errors:

One of the most common things, it is not as common in Java or Android, but anything using C or C++ - look for potential buffer overruns. And anything that has SQL Injections that do the same sorts of things: anything that can go outside of the expected bounds, that aren't being checked. And there are a number of Lint checkers that will pick up on that sort of thing. Use them! (P7)

Increasingly some of the reviewing features are being migrated from independent tools into the compilers and default build processes for mobile software:

So as tools get better, for both inspection and fixes, to say 'hey this might be a security flaw': as the compilers, as the development environment, whatever the tools are. Because even developers that are experts can make mistakes. And so the more the tools do like the code inspection review for you, for free, constantly, all the time, so you can't skip it, then yes, that will be a huge win. And I think that can instantly be improved in the two year time line. (P3)

Though of course there is little value to such warnings if the programmer ignores them:

Pay attention to the warnings, pay attention to the Link errors. [So it is not just the automated checks. It is the attitude towards those automated checks, taking them really seriously] Use them, don't forget them. (P7)

The tools need to be carefully designed to make them easy to use; Johnson et al. have researched a set of recommendations what is required [58]: in particular the ability to avoid repeated false positives and support for 'quick fixes'. Others have created developer support tools for Android app code security analysis: Xie et al. [111] were first; Nguyen et al. [77] support the more widely-used IntelliJ development environment..

Automated security testing comes in two forms. First is the automation of manual tests that have or could find security defects as automated regression tests, to avoid the risk that such defects may recur:

We added an entire section to [our automated testing suite] called 'Security', which is effectively hacking. We have built all form of vectored attacks against our platform – we endlessly think about ways to attack our platform. When we find a new one, we try to write a test for it. (P5)

However a second, recent, innovation is to use randomisation and ‘deep learning’ techniques to enable tests that would not necessarily occur to a human tester:

> *I actually find that our fuzzing efforts, which you could view as a form of code analysis, have quite a bit more tangible results. The fuzzing effort doesn't happen at code review time, but happens at check in time; we have clusters of machines where we are doing attacks against the software that is checked in, and we are able to find [exploits] very quickly. (P11)*

This approach is likely to be enhanced as tools and techniques develop:

> *So a deep learning system that actually understands security state machines that can look at code and not just see functionally what you are trying to do but have a look at the redundancy state machine, the security state machine, the parallelism and mass processing state machines, all those things as well. It can really help with actually saying ' you want a system to be like this but you don't really, you actually want your system to do this. And give you reasons why, as well. So that sort of guidance development, I think, is where we will be in the next 10 – 15 years. (P11)*

To get the best value, it is important to include both automated checks and automated testing as part of the fixed development process. Best practice, given that they are automated, is to include them within the build cycle.

> *Yeah, what we do is, [we have] a continual build system, every time someone checks in a change, we create a brand new version of [the system]. Once a day we snapshot that version ... into our testing infrastructure, and for that entire day we are doing attacks against the code that is running on that device. So next day a new version ..., and we continue attacks. And we will do that over and over again. (P11)*

7.5.4 Discussion

There is an art to using a code inspection tool effectively. Often developers are intimidated by a large number of warnings, many of which turn out to be spurious, in that what they are highlighting is not likely to cause a security issue. The recommendation of our interviewees was to work towards having a clean ‘compile’ such that only new problems show up.

> *There is another big one: when you compile something and it spits out a whole bunch of warnings – don't ignore them! That is something I have been really against for years, and watching the compilation of [a huge system] and all the warnings that it spits out – really bugs the crap out of me. It is a big job because they have left it so long that there are so many of them, but if they would just sit down and look through each and every warning, either say 'okay yes, I agree, it's a valid warning, in this case I know what I'm doing, it's okay so ignore it', or 'oh shit! I'd better take care of that!' Get rid of all warnings. (P7)*

> *[And to do that you need something in the tools that makes it possible to suppress a particular warning.] In C or C++ you there is #pragma. In Android Java you can have a link to an XML file that turns off certain lint features and you can also suppress warnings inline for specific items. (P7)*

Others, however, amongst my interviewees took the view that such tools add little value to a well written code base:

> *Traditionally, we haven't made strong use of Static Analysis tools. Static analysis tools have a reputation for being overly sensitive, and it's hard to find issues. ... I did an analysis of what [a commercial tool] found ... I wasn't able to find a real security vulnerability in all the stuff .... Mostly because it was either unreachable, or there was corruption but the corruption was such that it was not attacker controlled. At the time I was actually somewhat pro code analysis tools until [this analysis] forced it upon me, and then I became very negative towards them. (P11)*

The economics of building code inspection tools is also a little problematic. It is not in human nature to want to be proved wrong, and programmers are typically unwilling to go to the effort to find money for tools that do so:

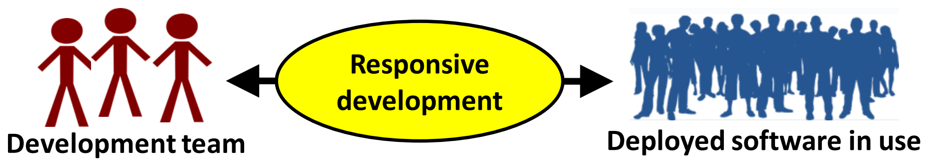
And critically [the industry needs] free automated tools. Paid tools are very hard to sell. How you get the economics right, so there is free automated tools end up out there is a hard problem. (P3)

In practice, companies like SonarSource [97] typically make their revenue from services, rather than the tools themselves.

Caution is needed with automated test tools, too. There is even a danger that an automated test suite, in the wrong hands, might itself become a threat:

You know, the guy who runs my security testing piece has said 'do you realise what we have built?' I said 'I absolutely know what we've built and that's why any execution of these is fully logged!' (P5)

7.6 Technique 6: Responsive Development



I think one of the problems with remote devices is that these devices are intended to be robust against all attackers if you lose your device... And that makes it challenging from a forensic point of view to look into [issues] (P11)

And the patches and updates basically what modern security is about – mistakes will be made and when the mistakes are found – how do you get the updates out? (P3)

7.6.1 Example

Jane Solo is planning her own personal app for the long-term. She knows she will need to change it and improve it, and that she will have defects to fix which will only appear in day-to-day activity by real users. To get feedback about those defects and enable her to take action, she puts in a good deal of logging. This uses standard third party libraries and back end services from companies that specialises in this. Using this library Jane instruments our code to give her feedback on which features are used most, she will get crash reports with details of where and how the error occurred; and she puts in a pop-up to users who have used her game for more than two weeks to ask them for feedback. Actually, the pop-up is more sophisticated and positive feedback replies go to the App Store; negative feedback goes to Jane as author!

The service itself ensures privacy, which will prevent Jane from getting some pieces of information that may be useful for tracking down bugs; Jane accepts this limitation in return for the convenience of using a well-made tool.

Jane also knows that the development of her product will not end so long as the product is live. She will continue to receive feedback of problems – including security issues – that have been identified, and she realises that as time goes on, the environment around her product changes and attackers get more sophisticated, there will be new exploits that she will have to defeat. Therefore, she plans a long-term program of continued product development, with releases at regular intervals. If ever she needs to stop supporting this program, then she will explicitly withdraw the product.

7.6.2 Exploration

With servers and cloud-based software the process of ensuring security is continuous; typically operators and even management will be keeping a close eye on what is happening from a security point of view to the system, and be prepared to take active action as a result.

> *I get an OSSEC [(an open-source monitoring system)] admin alert the minute anyone is trying to attack, and the great thing about OSSEC is it takes remedial action, moves them off, and we've got some other clever ideas we are thinking about. (P5)*

To keep apps secure also requires continuous feedback, both to detect actual exploits and to detect trends of use that may represent longer term threats. Getting such feedback is much more difficult with mobile apps than with servers. Not only are they not always connected, and under the control of someone else, but the devices are designed to be as impenetrable as possible:

> *[The OS designers] want to make sure that no matter whatever privileged position you have, that these devices are impenetrable. That is the goal. (P11)*

Responding to such feedback is also a continuous process. New exploits, improved processing power and wider publication of existing exploits all mean that what might have been secure a year ago may not be now.

> *Projects look at the risk here in their lifetime and you know the current risk and the current attack vectors, but they are constantly changing. (P8)*

> *It still is interesting to see how effectively security has a built in obsolescence. Even with SSL security, which is obviously almost the bottom level. (P12)*

The problem is not just increasing attack sophistication of attacks; changes to the supporting environment often have security implications requiring changes to apps to support them:

> *Obviously given the rate at which Apple and Google are changing Android and IOS and all the other things, just keeping still is difficult. (P12)*

However the nature of app development 'contracts', whether internal to a company or commercial external contracts is often 'fire and forget'.

> *[Most companies developing apps] treat the creation of what they do akin to building motorways or something – it's a project to deliver something, but that something is then just passed off to the highway authority for them to sit there and monitor the traffic flow on it, but aren't necessarily concerned that the bridge structure may not be up to the ever increasing amount of traffic that is passing over it right now. (P8)*

On completion of the initial app development phase, the development team is normally allocated to different projects.

> *Like many things that get delivered in a project, the project ends and interest dies with it. Unfortunately. And I think you lead into a significant challenge in securing things on an operational basis. (P8)*

This makes it very hard to pull together an ad-hoc team to solve even serious issues:

> *Technology is constantly changing but to bring together the spotlight or the focus on a live service, unless it has reached the stage that is it almost headline news, is very difficult to do because the effort required in creating a project in the first instance, to bring together the bodies and the budget for most businesses is enormous. So the day to day behaviour doesn't allow for the 'dipping into things'. (P8)*

Even given the development teams to analyse and fix software, ensuring that updates reach the users can also be a problem; many users do not enable automatic upgrades.

> *The moment you release something to an Android phone, you will, in general, never get a 100% update rate, because loads of people update software once and never update. (P3)*

7.6.3 Solution

Instigate a long-term development approach to support both security monitoring and regular updating. To achieve this, developers must find specific ways both to monitor feedback from the apps and to ensure the delivery of updates; and project stakeholders need to ensure that projects have a continuous long-term support and monitoring elements.

App feedback usually requires explicit functionality:

> *I've built quite a bit into the Apps where they have their own debug logs because I don't trust the likes of Google because they have to sanitise what they give you because they've got privacy issues on their side of things. Because we have more of a direct relationship with our users, we can get more information and we have them direct to our systems, so effectively there's a low level of logging, logging things which are going wrong. (P12)*

Typically this is not limited to security-based feedback, but can be enhanced to deliver security based information.

> *I must admit, most of the logging I tend to do is logging exceptions, well exceptions to the rule rather than Java exceptions: something funny has happened here, so you can say 'something's funny happened' (P12)*

Turning to the issue of acting upon the feedback, we identified two kinds of change a team needs to handle: longer term strengthening, and emergencies. The first requires regular releases of new software versions and a continuous resource to do so:

> *Part of my team's job is to make sure that the security issues that happen today, we eliminate them. There will be security issues that happen tomorrow, but they will be a different set of security issues. (P11)*

The second is an 'emergency', where a new exploit becomes public and the product needs an upgrade before one of the large number of unsophisticated ('script kiddie') attackers manages to use it on instances of the product:

> *What worries me more is script kiddies and things like that, because when you get a zero day exploit released and I must admit, that is the one time when I jump as quickly as possible because I think, if someone's just released an open SSL loophole or something, you can pretty much guarantee that within 24 hours someone will be probing every system they can find on the internet and they'll be breaking in. And that's not because they're coming after you personally. (P12)*

Getting the resource to do this requires a long-term approach to product development, since there will be costs long after the first release. Typically companies decide to maintain for a limited time and then explicitly stop security updates:

> *It involves engineering resource to do the ... updates across every product. What we have said is that the current, the products that are currently in this three year window [are maintained] so not everything [we have produced], but the current products, we will keep them up to date. (P3)*

Organisations taking this long-term attitude will use development contracts (‘maintenance contracts’) and system architectures that allow for this rather than the more traditional ‘fire and forget’ approach.

7.6.4 Discussion

Programmer feedback from live systems is an area where security and privacy needs can conflict. From a security point of view, it is valuable to have as much information as possible delivered back to the programmer. However from a privacy point of view it is not desirable to store personally identifiable information anywhere where it is not essential for the user’s needs. The simplest solution is to limit any code that creates the log entries, ensuring privacy-sensitive information is never logged at all. However this can conflict with debugging using test data: a programmer will want to see identifiable aspects of the test data in the logs. Therefore, another approach the author has used is to use structured log messages that identify sensitive data within the log messages, and to have the logging system remove the sensitive data only in the live system.

For mobile apps the upgrading process is usually straightforward for the developer; the ‘app store’/’play store’ and similar support relatively secure upgrading:

> *From my experience of being involved in both, not as a coder, but just being around the technology: [an advantage that apps have is] the fact that you can distribute an application through a trusted channel, with a high degree of confidence that is still going to be there in its intended form. (P8)*

However, as discussed above, there is a particular problem of ensuring that users upgrade to the new version. For iOS users this is typically not a major issue; anecdotal evidence suggests that typically 80% of users have upgraded most apps within three weeks [87]. However Android upgrade rates are slower; many users do not enable automatic upgrading – through ignorance, or because upgrading sometimes causes issues for apps. App statistics on this are hard to come by, but Thomas et al. obtained figures on the upgrades of the Android OS [103], finding:

> *Within 30 days of the first observation of a new version on a device, half of all devices of that model have the new version … installed, and within 324 days 95% of devices have the new version. (Thomas et al.)*

A common solution, implemented in several cases by the author, where the apps communicate with a server controlled by the app owner is ‘forced upgrades’ based on the app version number. This requires extra support in the app and server: on start-up the app interrogates the server for the minimum version currently supported; if the app’s current version is less, it refuses to run, and instead directs the user to the appropriate view in the ‘store’ app to make the upgrade.

Upgrading also carries with it its own security risks; the team will need to analyse these along with other risks:

> *There is a whole notion of trusted distribution as well, which is still very pertinent but which people have forgotten about. (P6)*

> *So when you update the firmware in a phone, technically speaking, you are attacking the phone. So there is a verification process involved there. (P2)*

> *Samsung had a massive problem with having their update mechanism open. So if you plugged a Samsung TV into your network and you are monitoring to find out Samsung connections, you could quite happily hack Samsung TVs globally. So that becomes a back door into your domestic network, an entry point, and away you go. (P10)*

If you do not have forced upgrades implemented, the need to support older versions of the app can become a major security issue since the whole system is only as strong as its weakest link:

> *I must admit, that’s difficult because you’ve got the risk of downgrade attacks with security, so if you have system which doesn’t use https, so you start off using http and then you put https in, you’ve got to be wary about turning it on, because there might be, it might not work. And if you’ve got a big user base, the last thing you want to find out is, you’ve just knackered half your user base. [So you need to be able to turn it on gradually] But the problem with that is that, because if you’ve got the ability to turn it on, a threat is the ability to turn it off and work around it, so you have to treat that as a transitory thing: you get to a stage where you have got everyone running secure and then you’ve got to disable the [http version]. (P12)*

There is also a range of architectural issues associated with upgrading, especially related to rollback and API versioning:

> *And also you cannot roll back perfectly. Because people don't take updates, but also because there is a corruption of user data issue going on, in that if, for example, someone installs Version 10 of the app, and modifies the local database to Version 10 format – if you try and roll back to Version 9, you might lose that data, and loose data protection, so rolling back is significantly harder in the mobile world. (P3)*

> *I must admit I did learn fairly quickly to make sure all your APIs are versioned properly so that [the code knows] when to fall back or reject them. And so you catch those things rather than it just falling over in a heap, because you're trying to do something where it's only got half the parameters and similar. (P12)*

8 Conclusion and Future Work

This chapter summarises the Dialectical Security techniques, explores parallels to existing literature, and discusses the contrast with both the patterns literature and conventional process-based approaches. It then reviews the experience of using Grounded Theory and examines how the findings of the study address the research questions introduced in Chapter 1, then explores threats to validity and possible future work to address them. Finally it discusses two areas for future research: exploring the techniques of Dialectical Security; and investigating a range of practical approaches to introduce them to developers.

8.1 Summary of Dialectical Security

Chapter 7 explored the experts' knowledge related to the research question:

RQ3 What are the most effective techniques to deliver app security?

It introduced six techniques of Dialectical Security, introducing each one with a specific example, then expanding it as a more general problem, offering a general solution, and discussing related issues and practical approaches. Table 6 below summarises the techniques.

Table 6: Summary of Dialectical Security Techniques

Technique	Summary
1. **Brainstorming the Enemy**	Use ideation sessions with fellow programmers and others to identify both attackers and possible exploits in two steps
2. **Negotiated Security**	Interpret the security risks and costs to project stakeholders in terms they can understand and use to prioritise security concerns against other organisation and project needs.
3. **Cross-Team Security Discussion**	Ensure frequent and open communication on security problems between development teams in any way available.
4. **Security Challenge**	Set up the development so that each has another person or team with a different viewpoint challenging the security and privacy aspect of assumptions, decisions and code.
5. **Automated Challenge**	Use automated code analysis, and automated security testing to create dialectical challenges to the programmers.
6. **Responsive Development**	Instigate a long-term development approach to support both security monitoring and regular updating of the apps.

8.2 Discussion of Dialectical Security

The ordering of these six techniques is roughly chronological from the point of view of the development team. While each is used repeatedly throughout the development cycle, developers will encounter the need for Brainstorming the Enemy and Negotiated Security earlier in each development cycle; and Responsive Development naturally comes rather later.

8.2.1 Relationship to Existing Work

We identified these techniques through the Grounded Theory process applied to our interview data; afterwards we found parallels in existing research, as follows.

Comparing the existing work on app developers and security, section 3.4 identified valuable research on current practice by developers such as that by Balebako et al. [10]; this work goes further by identifying approaches for *better* practice. Much of the remaining literature we discussed is also valuable in the context of Dialectical Security as providing solutions to the challenges identified through dialectic. Thus an Android app programmer who is made aware of security issues from Brainstorming the Enemy, Security Challenge or Automatic would then be motivated to search for solutions on the web [120] or in practitioners' literature such as 'Android Security Internals' [38]; work by Acar et al. suggests that their best choice would be the literature [2].

Considering the Dialectical Security techniques themselves, we suggest that two are reasonably well-understood and researched in various ways: Security Challenge and Automated Challenge. The techniques of Security Challenge of reviews and penetration testing are explored in detail in literature; Responsive Development is novel in the app development context, but the techniques of continuous response to security challenges are well-known within the context of server system management. McGraw's book [68], for example, discusses all of these. For Automated Challenge, there is a considerable range of automated validation tools available even if, as found by Johnson et al. [58], these are currently not often used by developers.

The other three techniques, Brainstorming the Enemy, Negotiated Security and Cross-Team Security Discussion are less well reflected in existing security literature; section 8.7 proposes approaches to research them. However they do have parallels in other aspects of software engineering. Brainstorming the Enemy relates to the HCI concept of 'personas' [86]; Negotiated Security relates in to the agile 'Planning Game' technique [14]; and Cross-team Security Discussion relates to the large amount of work available on collaboration between distributed teams [20].

An important piece of related work, published after the main work of this thesis, is by Ashenden and Lawrence [9]. They used an Action Research approach to investigate and improve the relationships between security professionals and software developers. The Action Research approach has considerable potential, and the work suggests an important further 'dialectical technique' (surprisingly, one not mentioned by our interviewees) in the interaction between programmers and security professionals themselves.

8.2.2 From Processes to Dialectic Cultures

Section 1.1 identified that existing literature contains little about the team interactions required to achieve software security. The Dialectical Software techniques by contrast constitute a way of working, almost an attitude to working, for developers who need to deliver secure software. They are completely consistent with, and incorporate the thinking of, much existing literature, but extend it to provide immediate day-to-day help to developers.

Where they differ from existing literature is in their implied approach to team organisation. Secure Development Processes like Microsoft's [69] provide a series of steps and deliverables for a team to carry out. Dialectical Software instead provides a set of attitudes to development teamwork and approach, and therefore meshes more effectively with self-organising teams [53].

Because it is interactive, and an attitude of mind more than a formal method that needs to be followed, we propose that Dialectical Security will also appeal more to app developers than many of the existing approaches.

8.2.3 Patterns of a Different Kind

As discussed in section 6.2, the format used to describe the Dialectical Security techniques is based on the Design Patterns format [46]. The format works particularly well because of several aspects: the name makes each item easy to discuss and remember; the repeated structure makes them easy to follow (chapter 4 showed that this approach is now used by many non-pattern books); and the implied problem-context-solution format helps readers to decide whether the technique is appropriate to particularly situations.

The main difference from the design patterns format is that where patterns gain authority from 'known uses', these techniques take their authority from the Grounded Theory analysis, grounded in existing practice and substantiated by quotations; we believe this makes them more compelling than, for example, Schumacher et al.'s 'security process' patterns [94].

8.3 Experience of Grounded Theory

The analysis followed the lines outlined in Section 2.5 'Grounded Theory Step-by-Step'. As a newcomer to Grounded Theory, the author was surprised to find that the approach worked opportunistically – in that each new step addressed a problem that he had gradually identified in carrying out the earlier steps. So for example, the categorisation phase addressed the problem 'how do we locate existing codes now that we've got so many?' Then as he started an initial write-up of one of the conclusions, he encountered a new problem 'the data points us to this conclusion; how, objectively, can we back up or reject that conclusion?' Core Categories addressed that problem. Lastly, in writing up, he needed to create a compelling narrative and to justify specific theories that arose from the analysis; sorting addressed that problem.

8.4 Revisiting Objectives

In the introduction to this thesis we discussed the research question:

RQ1 What techniques and ideas will appeal to development teams and lead to them developing more secure app software?

That led to three other questions, about how the experts themselves were motivated and how they learned; about the most effective techniques to deliver app security; and about ways of introducing those techniques to developers and teams. The rest of this section looks at each question in turn.

RQ2 What motivated the experts themselves to learn software security; how did they do so; and how do they continue to learn?

Chapter 5 considered this in some detail. We explored four forces affecting developers' motivation: knowledge, tasks, worry and enthusiasm. Our conclusion was that the experts' own motivations were mainly due to enthusiasm, and they had generally picked up their knowledge – and continued to learn – on the job and through hobby work rather than through any kind of formal instruction or learning. Our novel finding related to this is that most app developers, by contrast with the experts, have little knowledge or even interest related to app security.

RQ3 What are the most effective techniques to deliver app security?

Chapter 6 outlined six techniques of 'Dialectic Security', which encapsulate aspects of good app security practice highlighted by the experts we interviewed. While we do not have evidence to state objectively that these are *the most effective* techniques, we can be sure that they are *effective* techniques, and that our interviewees considered them to be amongst the most useful available to them. The novel finding is that these techniques relate not to the artefacts produced, nor to formal Secure Development Processes, but rather to a culture of encouraging challenges from a variety of counterparties – a culture of 'dialectic'.

RQ4 How should we effectively introduce security to app development teams?

Chapter 5 outlines the opinions of the experts interviewed on appropriate ways to motivate and teach development teams. These, though, varied very considerably, leading us to conclude that the discipline of app development security is at an early stage. We shall revisit this question in section 8.8.1, outlining possible approaches and ways to evaluate them.

8.5 Research Validity and Verifiability

How certain can we be that this theory accurately reflects reality? We approach this question by analysing threats to validity.

Considering first Conclusion Validity, do the research data justify the conclusions? Grounded Theory's rigorous process of line by line coding, categorisation, and sorting generates theory that does reflect the interview data. The use of extensive quotations ensures that this can be at least partially checked.

In terms of Construct Validity, does the Dialectical Security theory represent actual practice? GT handles this primarily in terms of 'theoretical saturation', reached when new interviews do not add substantially to the theory. Guest [51] suggests that a dozen interviews are often sufficient for this; in this case as researchers we believe we have reached theoretical saturation with regard to the list of techniques, but not with regard to all the potential detail to be uncovered within each technique. There is also a risk of bias in the choice of interviewees, and of questions; we addressed this with interviewees from a wide range of industry roles, and completely open questions [109].

In terms of External Validity, can the results be generalised to a wider scope? GT's conclusions are always limited to the specific scope studied [21]. In this case since many of the experts were familiar with – and sometimes describing – more general secure software development, some conclusions will apply to non-app development. We can however make no claims of applicability to different development cultures other than UK and US-based companies.

Finally we should qualify what validity we are discussing. The interview process has determined industry understanding of best practice (community knowledge of 'truth'); this may possibly not correspond to actual best practice (objective 'truth'). The Testability discussion in section 8.5.1 and the research suggested in section 8.7 address this limitation.

8.5.1 Verifiability

We propose two approaches to verify this theory:

Repeatability: First, an independent researcher can join the team (hence preserving confidentially) to reanalyse the existing transcriptions to validate or challenge the Conclusion Validity. Second, we can return to those interviewees who consent, to explore aspects of the techniques in more detail. Third, we, or a different research team, may repeat the GT-based interviews with a different set of experts to explore if the theory derived is consistent with Dialectical Security – or if it extends to different development cultures.

Testability: The theory implies that introducing Dialectical Security techniques will improve app security. The authors' paper 'Reaching the Masses' [107] explores approaches to introduce such techniques and evaluate whether this improvement does happen.

8.6 Proposals for Future Work

We have identified two further areas for future work, exploring the research questions RQ3 and RQ4 respectively: research to expand knowledge of the Dialectical Security techniques; and research to discover ways of introducing them to app developers. The next two sections examine them in detail.

8.7 Researching Dialectical Security Techniques

Section 8.1 identified that three of the techniques, Security Challenge, Automated Challenge and Responsive Development, are relatively well-understood. Therefore we propose that further research examine the three less well understood techniques: Brainstorming the Enemy, Negotiated Security and Cross-team Security Discussion. We suggest proposed research questions along the lines of

PRQ1 What are the most effective ways to ideate understanding of attackers and potential exploits?

PRQ2 How best do we represent security questions in business terms?

PRQ3 What forms of cross-team interaction are most effective to ensure app security?

There are several possible approaches to this research. Experimental approaches might trial a variety of representations of security questions with a number of product managers; or set up different groups of Computer Science students with different ideation techniques and compare their success at identifying attackers and exploits. An ethnographic approach might follow the progress of a development team, identifying where the major security mitigations were identified and how the negotiations took place in practice. A survey approach, by contrast, might ask the questions of a variety of developers and stakeholders to produce a possible consensus.

8.8 Researching Teaching Interventions

Section 5.1 identified that few developers are knowledgeable or even motivated to improve app security. To improve the situation we need to reach out to a group of individuals, without having direct access to them or direct influence on them. We need a new paradigm; we need a new way to reach these people.

8.8.1 A Different Approach

Different programmers learn in different ways and are interested in different things, so we believe a single form of intervention, however effective, is unlikely to reach all of our target audience. In addition, since we are in effect teaching new attitudes, few of the traditional mechanisms such as books are likely to work.

We propose instead ‘engaging’ interventions likely to appeal to programmers for their own sake. We anticipate that these will be publicised via the web: expert blogs, and security OS websites.

The following sections explore some possibilities for these interventions.

8.8.2 Games That Teach

One popular approach is games. A great deal of work has been done on gamification, with books such as Kapp’s [62] explaining the techniques involved. Tillman et al.’s game Code Hunt [104] teaches vast numbers of programmers through an online game. Code Hunt’s approach is to provide a unit test that the programmer’s code must pass; this certainly demonstrates the dialectic aspect, but will not be very good for teaching the other security techniques. Other researchers, including the authors, have had success with group games to teach aspects of software security, such as Denning et al.’s Control-Alt-Hack game [34]. These work very well in a classroom or conference context, but do not naturally extend to reach to an online audience.

Instead we suggest solo or multiplayer games suitable for distributed players. Picture Angry Birds meeting Stack Overflow! Have players implement security aspects to defend against attacks? Perhaps crowd source both attacks and defences, where each player gets to both take the role of attacker on other players’ code, and defender on their own. It is an enchanting possibility; even if it risks taking too much time to engage the typical target solo programmer.

8.8.3 Story Telling

A different approach is story-telling. The British radio soap opera, The Archers, has been running for 65 years, and has over 5 million regular listeners; its main purpose, at which it is highly successful, is to teach farming knowledge to a community that is unreachable by any other form of education. Taking a similar approach here would suggest a podcast (and blog) narrating a plot that would cover and teach each of these aspects.

More ambitious would be a storyline in an appropriate existing series (‘Mr. Robot’, and the UK’s ‘IT Crowd’ come to mind), to be created if the opportunity arose.

A related approach might be through a comic strip already popular with programmers, such as the XKCD series [88]; the back archives of such comic strips would give the benefit of something permanent, easily accessible and shareable by developers. Zhang-Kennedy et al. used such an approach with success to teach security in the context of end users [117].

8.8.4 Adapting Business as Usual Approaches

More conventional is to tailor direct teaching and group learning approaches to the distributed nature of the target audience. This suggests implementing a massively open online course (MOOC) on app security using audio, written text, and video along with interactive discussion groups. Organisations such as edX and Futurelearn provide frameworks to make this straightforward [122,123].

Another possibility is a short video along the lines of – or indeed actually – a TED talk by a suitable expert.

Both possibilities leverage the 'professional skills gaining' motivation present in programmers, which suggests promoting them via professional organisations too.

8.8.5 Research Agenda

Whilst each of these interventions has promise, we do not know which are likely to be effective, nor which techniques and variants of each will have the most impact. This leads us to a set of proposed research questions:

PRQ4 How best to design and implement the interventions to convey the Dialectical Security techniques? This is a complex problem, involving amongst other aspects elements of design, gamification and measurement of impact.

PRQ5 Which interventions – and dissemination techniques – are most effective at conveying each technique to the largest population of programmers? Implementing all the interventions at scale will be costly; we shall need to evaluate which ones offer the most value.

PRQ6	Which interventions provoke a wider interest in the programmers reached? To achieve a lasting effect we do not just need to engage programmers initially, but need also to encourage further interest and learning in the subject.

This approach is very different from others in the field of programmer education, making this an entirely new subdiscipline. The research will require a multi-disciplinary team, with varying skills, including at least the following:

Programming:	To implement code based interventions such as games.
Psychology:	To achieve the ‘attractiveness’ of the content; to structure measurement of the results; to use psychological techniques to ‘nudge’ programmers towards more effective security practices.
Creative writing:	For the storyline.
Narration:	For an engaging verbal version of the storyline.
Marketing:	To establish and develop the channels to bring the content to the target audience.

8.8.6 Evaluating Techniques

The research will require objective measurement. In particular we can identify four aspects to measure:

Success	Using the interventions with a sample group of students or similar, and evaluating their learning based on the intervention (PRQ4).
Reach	The number of downloads, accesses, or to the resource (PRQ5)
Engagement	The number of accesses of later parts of the resource. (PRQ5)
Coverage	Attending exhibitions such as Apps World frequented by the target solo programmer audience and asking via a simple questionnaire of delegates which if any of the interventions they have encountered and their impact (PRQ5 PRQ6).

Ideally we shall want to extend our research to measure outcomes as well as these outputs. Whilst we can argue that the combination of ‘success’ with ‘engagement’ and ‘coverage’ implies a positive impact, better still would be evidence of an improvement in the code produced by programmers in the target group.

To achieve that, we might collect app identifiers, where possible, from participants for a ‘before and after’ anonymous evaluation of their released apps’ security along the lines of that by Enck et al. [39]. Other possibilities would include extending the questionnaires in the ‘coverage’ evaluation to estimate interviewees’ awareness of app security, and correlating that with exposure to the interventions.

8.9 Conclusion

To summarise, in this Grounded Theory study using interviews of experts in secure app development, we encountered three particular surprises. First was a clear indication that the discipline of app security is at a very early stage of development. Second was a significant discrepancy between current industry understanding of the approach required by app developers, and the experts’ recommendations of good practice. Finally we found that some of the best techniques for software security were not in terms of artefacts and reports, nor formal processes, but a culture in the developers themselves.

The study generated a theory of ‘Dialectic Security’, continuing challenging dialogue with different counterparties; chapter 6 explores six techniques within this theory. We conclude that these techniques are well-suited for app development teams in the majority of organisations. We can investigate the techniques further as discussed in section 8.7; we can also look for ways to disseminate them more widely, and research interventions as discussed in section 8.8 to introduce them into a range of existing development teams.

Using these techniques, we believe, will enhance the future security of apps, and lead to better safety for all of those who use them.

References

[1] Acar, Y., Backes, M., Bugiel, S., Fahl, S., Mcdaniel, P.D., and Smith, M. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016,* (2016), 433–451.

[2] Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L., and Stransky, C. You Get Where You're Looking For. *IEEE Symposium on Security and Privacy,* (2016), 289–305.

[3] Adolph, S., Hall, W., and Kruchten, P. Using Grounded Theory to Study the Experience of Software Development. *Empirical Software Engineering 16,* 4 (2011), 487–513.

[4] Akerlof, G.A. The Market for "Lemons": Quality Uncertainty and the Market Mechanism. *Quarterly Journal of Economics 84,* 3 (1970), 488–500.

[5] Alexander, C. *The Timeless Way of Building*. New York: Oxford University Press, 1979.

[6] Allan, G. A Critique of Using Grounded Theory as a Research Method. *The Electronic Journal of Business Research Methods 2,* 1 (2003), 1–10.

[7] Anderson, R. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2008.

[8] Apple. Introduction to Secure Coding Guide. https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html.

[9] Ashenden, D. and Lawrence, D. Security Dialogues : Building Better Relationships. *IEEE Security & Privacy Magazine*, June (2016).

[10] Balebako, R., Marsh, A., Lin, J., Hong, J., and Cranor, L. The Privacy and Security Behaviors of Smartphone App Developers. *Internet Society*, October (2014).

[11] Banks, A. and Edge, C.S. *Learning iOS Security*. Packt Publishing, Birmingham, UK, 2015.

[12] Barua, A., Thomas, S.W., and Hassan, A.E. *What Are Developers Talking about? An Analysis of Topics and Trends in Stack Overflow*. 2012.

[13] Baum, T., Liskin, O., Niklas, K., and Schneider, K. Factors Influencing Code Review Processes in Industry. *FSE2016*, (2016).

[14] Beck, K. and Fowler, M. *Planning Extreme Programming*. Addison-Wesley Professional, 2001.

[15] Beecham, S., Baddoo, N., and Hall, T. Motivation in Software Engineering : A Systematic Literature Review. *Information and Software Technology 50*, 9 (2008), 860–878.

[16] Bejtlich, R. Reviews of Six Software Security Books. 2006. http://taosecurity.blogspot.co.uk/2006/11/reviews-of-six-software-security-books.html.

[17] Blackwell, C. and Zhu, H. *Cyberpatterns*. Springer, Heidelberg New York Dordrecht London, 2014.

[18] Bluebox Security. *'Tis the Season to Risk Mobile App Payments - An Evaluation of Top Payment Apps*. 2015.

[19] Bruce Schneier. Schneier on Security: Crypto-Gram.

https://www.schneier.com/crypto-gram/.

[20] Carmel, E. *Global Software Teams: Collaborating across Borders and Time Zones*. Prentice Hall PTR, 1999.

[21] Charmaz, K. *Constructing Grounded Theory*. Sage, London, 2014.

[22] Chell, D., Erasmus, T., Colley, S., and Whitehouse, O. *The Mobile Application Hacker's Handbook*. John Wiley & Sons, Indianapolis, 2015.

[23] Chevalier, J. and Goolsbee, A. Measuring Prices and Price Competition Online: Amazon.com and BarnesandNoble.com. *Quantitative Marketing and Economics 1*, 2 (2003), 203–222.

[24] Clarke, S. What Is an End User Software Engineer? *Dagstuhl Seminar Proceedings (07081 - End-User Software Engineering)*, (2007), 1–2.

[25] Cockburn, A. and Williams, L. The Costs and Benefits of Pair Programming. In *Extreme Programming Examined*. 2001, 223–243.

[26] Conradi, R. and Dybå, T. An Empirical Study on the Utility of Formal Routines to Transfer Knowledge and Experience. *ACM SIGSOFT Software Engineering Notes 26*, 5 (2001), 268–276.

[27] Cooperrider, D.L. and Whitney, D. Appreciative Inquiry: A Positive Revolution in Change. *Appreciative Inquiry*, (2005), 30.

[28] Cooperrider, D.L., Whitney, D.K., and Stavros, J.M. *Appreciative Inquiry Handbook*. Berrett-Koehler Publishers, 2003.

[29] Cravens, A. A Demographic and Business Model Analysis of Today's App Developer. *GigaOM Pro, September*, (2012).

[30] Creswell, J.W. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage publications, 2013.

[31] Dai Zovi, D.A. Apple iOS 4 Security Evaluation. *BlackHat USA*, 2011. http://media.blackhat.com/bh-us-11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf.

[32] Dashti, M.T. and Basin, D. Security Testing Beyond Functional Tests. *Engineering Secure Software and Systems*, Springer (2016), 1–19.

[33] DeMarco, T. and Lister, T. *Peopleware: Productive Projects and Teams*. Addison-Wesley, NJ, 2013.

[34] Denning, T., Lerner, A., Shostack, A., and Kohno, T. Control-Alt-Hack: The Design and Evaluation of a Card Game for Computer Security Awareness and Education. *CCS '13: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, (2013), 915–928.

[35] Drake, J.J., Lanier, Z., Mulliner, C., Fora, P.O., Ridley, S.A., and Wicherski, G. *Android Hacker's Handbook*. John Wiley & Sons, Indianapolis, 2014.

[36] Dybå, T. An Empirical Investigation of the Key Factors for Success in Software Process Improvement. *IEEE Transactions on Software Engineering 31*, 5 (2005), 410–424.

[37] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An Empirical Study of Cryptographic Misuse in Android Applications. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*, (2013), 73–84.

[38] Elenkov, N. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, San Francisco, 2014.

[39] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. *Proceedings of the 20th USENIX Conference on Security*, (2011).

[40] Enes, P. and Conradi, R. Acquiring and Sharing Expert Knowledge. 2005. http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2005/aanes-fordyp05.pdf.

[41] Enisa. Smartphone Secure Development Guidelines for App Developers. *Enisa*, (2011), 17.

[42] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben,

B. Why Eve and Mallory Love Android : An Analysis of Android SSL Security Categories and Subject Descriptors. *Proceedings of the 2012 ACM Conference on Computer and Communications Security - CCS '12,* ACM Press (2012).

[43] Faily, S. and Flechais, I. Persona Cases: A Technique for Grounding Personas. *Chi 2011*, (2011), 2267–2270.

[44] Fisher, R., Ury, W.L., and Patton, B. *Getting to Yes: Negotiating Agreement Without Giving In*. Penguin, 2011.

[45] Furniss, D., Blandford, A.A., and Curzon, P. Confessions from a Grounded Theory PhD: Experiences and Lesson Learnt. *Proceedings of the 2011 Annual Conference on Human Factors in Computing Systems - CHI '11,* (2011), 113.

[46] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[47] Glaser, B.G. and Strauss, A.L. *The Discovery of Grounded Theory : Strategies for Qualitative Research*. Aldine Transaction, Chicago, 1973.

[48] Glaser, B.G. *Theoretical Sensitivity*. Sociology Press, 1978.

[49] Gollmann, D. *Computer Security*. Chichester : Wiley, 2011.

[50] Google. Android Security Tips. http://developer.android.com/training/articles/security-tips.html.

[51] Guest, G., Bunce, A., and Johnson, L. How Many Interviews Are Enough? An Experiment with Data Saturation and Variability. *Field Methods 18,* 1 (2006), 59–82.

[52] Hafiz, M., Adamczyk, P., and Johnson, R.E. Growing a Pattern Language (for Security). *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! '12,* (2012), 139.

[53] Hoda, R., Noble, J., and Marshall, S. Organizing Self-Organizing Teams. *Proceedings of the 32nd ACM/IEEE International Conference on Software*

Engineering (ICSE '10) - Volume 1, (2010), 285–294.

[54] Hoda, R., Noble, J., and Marshall, S. Grounded Theory for Geeks. *Conference on Pattern Languages of Programs*, ACM (2011), 1–17.

[55] Howard, M., LeBlanc, D., and Viega, J. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, Inc., 2009.

[56] ISO/IEC. ISO/IEC 21827:2008 - Systems Security Engineering - Capability Maturity Model. *2008*, (2008), 144.

[57] Jackson, M., Crouch, S., and Baxter, R. Software Evaluation: Criteria-Based Assessment. *Software Sustainability Institute, ...*, (2011), 1–13.

[58] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? *2013 35th International Conference on Software Engineering (ICSE)*, IEEE (2013), 672–681.

[59] Johnson, M. and Senges, M. Learning to Be a Programmer in a Complex Organization. *Journal of Workplace Learning 22*, 3 (2010), 180–194.

[60] Judge, S. Android App Security. http://www.androidsecurity.guru.

[61] Judge, S. Private Communication. 2016.

[62] Kapp, K.M. *The Gamification of Learning and Instruction: Game-Based Methods and Strategies for Training and Education*. John Wiley & Sons, San Francisco, 2012.

[63] Kienzle, D.M., Elder, M.C., Tyree, D., and Edwards-Hewitt, J. Security Patterns Repository Version 1.0. *DARPA, Washington DC*, (2002).

[64] Komatineni, S. and MacLean, D. *Pro Android 4*. Apress, 2012.

[65] Lerch, J., Hermann, B., Bodden, E., and Mezini, M. FlowTwist: Efficient Context-Sensitive Inside-out Taint Analysis for Large Codebases. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (2014), 98–108.

[66] LLVM Project. libFuzzer. http://llvm.org/docs/LibFuzzer.html.

[67] Makan, K. and Alexander-Bown, S. *Android Security Cookbook*. Packt Publishing Ltd, 2013.

[68] McGraw, G. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[69] Microsoft. Microsoft Secure Development Lifecycle. https://www.microsoft.com/en-us/sdl/.

[70] Microsoft. Learning Security - MSDN. https://msdn.microsoft.com/en-us/security/aa570420.aspx.

[71] Munawar Hafiz. Security Pattern Catalog. http://www.munawarhafiz.com/securitypatterncatalog/index.php.

[72] Murphy-Hill, E., Lee, D.Y., Murphy, G.C., and McGrenere, J. How Do Users Discover New Tools in Software Development and Beyond? *Computer Supported Cooperative Work (CSCW) 24*, 5 (2015), 389–422.

[73] Myers, G.J., Sandler, C., and Badgett, T. *The Art of Software Testing*. John Wiley & Sons, 2011.

[74] Nadi, S., Krüger, S., Mezini, M., and Bodden, E. Jumping Through Hoops : Why Do Java Developers Struggle With Cryptography APIs ? *ICSE16: 38th IEEE International Conference on Software Engineering*, (2015).

[75] Naqvi, S.A.A. The Grounded Incident Fault Theories (GIFTs) Method. 2014.

[76] Near, J.P. and Jackson, D. Finding Security Bugs in Web Applications Using a Catalog of Access Control Patterns. *Proceedings of the 38th International Conference on Software Engineering*, ACM (2016), 947–958.

[77] Nguyen, D., Acar, Y., and Backes, M. *Developers Are Users Too : Helping Developers Write Privacy Preserving and Secure (Android) Code*. 2016.

[78] Noble, J. and Weir, C. *Small Memory Software: Patterns for Systems with Limited Memory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA,

USA, 2001.

[79] Oates, B.J. *Researching Information Systems and Computing*. 2006.

[80] Ben Othmane, L., Ranchal, R., Fernando, R., Bhargava, B., and Bodden, E. Incorporating Attacker Capabilities in Risk Estimation and Mitigation. *Computers & Security 51*, (2015), 41–61.

[81] OWASP Foundation. *OWASP Code Review Guide Book*. OWASP Foundation, 2008.

[82] Pfleeger, C.P. and Pfleeger, S.L. *Security in Computing*. Prentice Hall Professional Technical Reference, 2002.

[83] Pirsig, R.M. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. Random House, 1999.

[84] Ponemon Institute. *The State of Mobile Application Insecurity*. 2015.

[85] Proksch, S., Bauer, V., and Murphy, G.C. How to Build a Recommendation System for Software Engineering. In *Software Engineering - International Summer Schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures*. 2014, 1–42.

[86] Pruit, J. and Grudin, J. Personas: Practice and Theory. *Proceedings of the 2003 Conference on Designing for User Experiences*, ACM (2003), 1–15.

[87] Quora. How Frequently Do Users Actually Update Their iOS Apps? https://www.quora.com/How-frequently-do-users-actually-update-their-iOS-apps.

[88] Randall Munroe. XKCD: A Webcomic of Romance, Sarcasm, Math and Language. http://xkcd.com/.

[89] Reed, J. *Appreciative Inquiry: Research for Change*. Sage, 2006.

[90] Rigby, P.C. and Bird, C. Convergent Contemporary Software Peer Review Practices. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, (2013), 202.

[91] Romanosky, S. Security Design Patterns Part 1. *Proceedings of PLoP*, (2001), 1–19.

[92] SANS Institute. SANS Institute Security Resources. https://www.sans.org/security-resources/.

[93] Schneier, B. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2011.

[94] Schumacher, M., Fernandez-buglioni, E., Hybertson, D., Buschmann, F., and Sommerlad, P. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[95] Shih, Patrick C. and Venolia, Gina and Olson, G.M. Brainstorming Under Constraints: Why Software Developers Brainstorm in Gro Ups. *Proceedings of the 25th BCS Conference on Human-Computer Interaction*, (2011), 74–83.

[96] Six, J. *Application Security for the Android Platform*. O'Reilly, Sebastapol, CA, 2011.

[97] SonarSource SA. Sonar Code Inspection. http://www.sonarqube.org.

[98] Steel, C., Nagappan, R., and Lai, R. *Core Security Patterns*. Prentice Hall, 2006.

[99] Sterling, G.D. and Brinthaupt, T.M. Faculty and Industry Conceptions of Successful Computer Programmers. *Journal of Information Systems Education 14*, 4 (2003), 417.

[100] Stol, K., Ralph, P., and Fitzgerald, B. Grounded Theory in Software Engineering Research : A Critical Review and Guidelines. *Proceedings of the 38th International Conference on Software Engineering*, ACM (2015), 120–131.

[101] Strauss, A.L. and Corbin, J.M. *Basics of Qualitative Research*. Sage Newbury Park, CA, 1990.

[102] The Allium. Computer Programming To Be Officially Renamed "Googling

Stackoverflow.” http://www.theallium.com/engineering/computer-programming-to-be-officially-renamed-googling-stackoverflow/.

[103] Thomas, D., Beresford, A., and Rice, A. Security Metrics for the Android Ecosystem. *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, (2015), 87–98.

[104] Tillmann, N., de Halleux, J., Xie, T., and Bishop, J. Code Hunt: Gamifying Teaching and Learning of Computer Science at Scale. *Proceedings of the First ACM Conference on Learning@ Scale Conference*, ACM (2014), 221–222.

[105] Vidas, T., Cylab, E.C.E., Votipka, D., Cylab, I.N.I., and Christin, N. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. *WOOT*, (2011), 81–90.

[106] Vision Mobile. *Developer Economics Q3 2014: State of the Developer Nation*. London, 2014.

[107] Weir, C., Rashid, A., and Noble, J. Reaching the Masses: A New Subdiscipline of App Programmer Education. *FSE’16: 24nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering Proceedings: Visions and Reflections*, ACM (2016).

[108] Weir, C. Penrillian’s Secure Development Process. 2013. http://www.penrillian.com/sites/default/files/documents/Secure_Development_Process.pdf.

[109] Weir, C. How to Improve the Security Skills of Mobile App Developers: Comparing and Contrasting Expert Views. 2016.

[110] Wikipedia. Dialectic. https://en.wikipedia.org/wiki/Dialectic.

[111] Xie, J., Chu, B., Lipford, H.R., and Melton, J.T. ASIDE: IDE Support for Web Application Security. *Proceedings of the 27th Annual Computer Security Applications Conference on - ACSAC ’11*, (2011), 267.

[112] Xie, J., Lipford, H.R., and Chu, B. Why Do Programmers Make Security Errors? *Proceedings - 2011 IEEE Symposium on Visual Languages and Human*

Centric Computing, VL/HCC 2011, (2011), 161–164.

[113] Yoder, J. and Barcalow, J. Architectural Patterns for Enabling Application Security. *Proceedings of PLoP 1997*, (1998), 31.

[114] Yskout, K., Heyman, T., Scandariato, R., and Joosen, W. *A System of Security Patterns*. Heverlee, 2006.

[115] Yskout, K., Heyman, T., Scandariato, R., and Joosen, W. *Security Patterns: 10 Years Later*. Heverlee, 2008.

[116] Yskout, K., Scandariato, R., and Joosen, W. Do Security Patterns Really Help Designers? *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE (2015), 292–302.

[117] Zhang-Kennedy, L., Chiasson, S., and Biddle, R. The Role of Instructional Design in Persuasion: A Comics Approach for Improving Cybersecurity. *International Journal of Human-Computer Interaction 32*, 3 (2016), 215–257.

[118] Stack Overflow Developer Survey 2016 Results. http://stackoverflow.com/research/developer-survey-2016#developer-profile.

[119] OWASP Developer Guide. https://github.com/OWASP/DevGuide.

[120] OWASP Mobile Security Project - Top Ten Mobile Risks. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.

[121] William Brandon - Private Communication. .

[122] edX. https://www.edx.org.

[123] Futurelearn. https://www.futurelearn.com.

Appendices

Diagram of Key Codings

The following diagram shows a subset of the GT codings from the interviews, arranged to show the relative frequency of coding. This represents a ‘sorted’ set of codes – merged and recreated from the original coding. The full names of the codes are given in the following section.

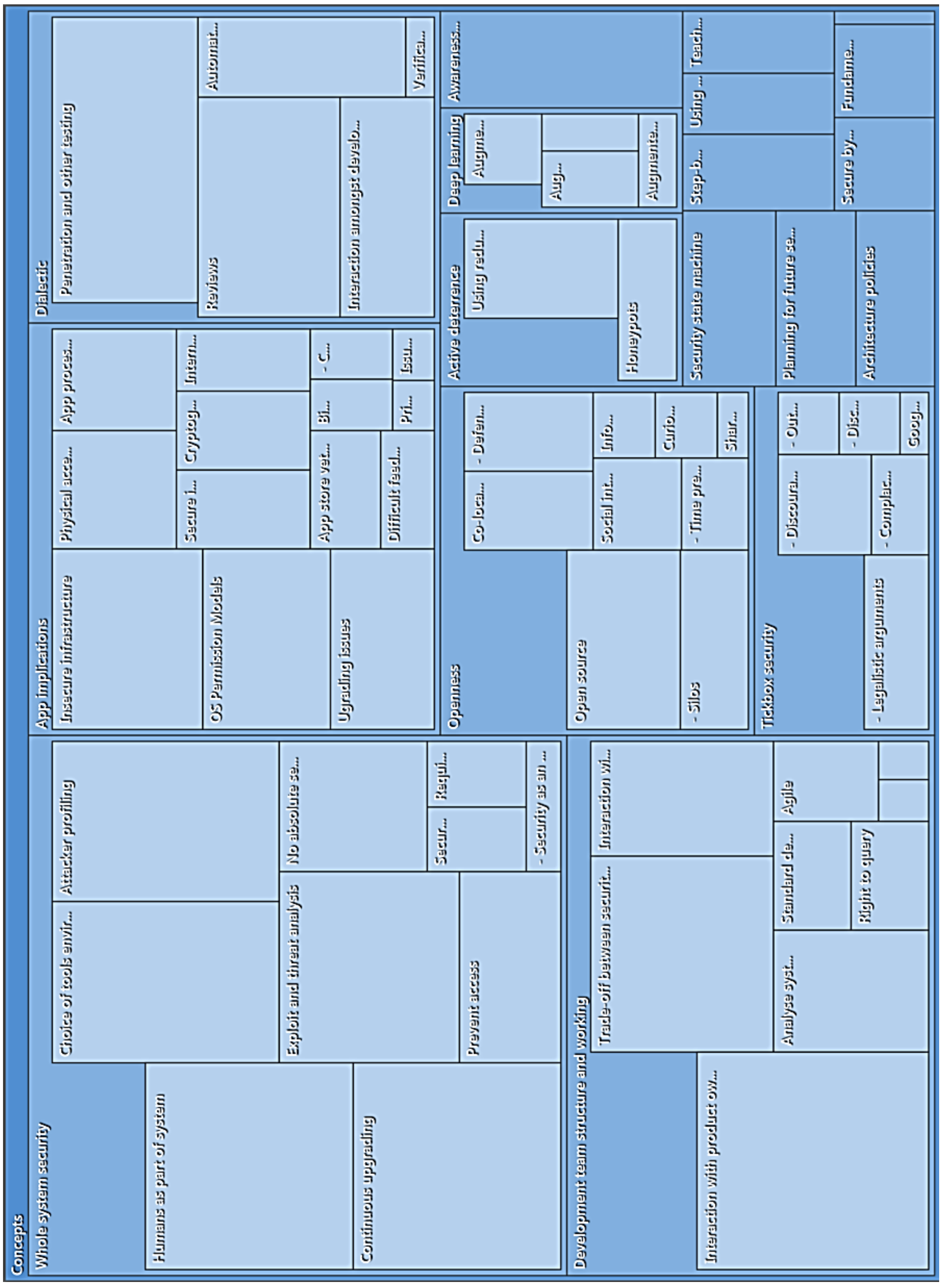


Table of Key Codes

The table below shows key Grounded Theory codes, along with the number of interviews in which they were found (Srcs) and the number of times each was referenced within those interviews (Refs).

Codes	Srcs	Refs
Concepts	0	0
Active deterrence	4	18
Honeypots	2	5
Using redundancy	3	8
App implications	0	0
- Can spoof	2	2
App processing and storage benefit	4	6
App store vetting	3	4
Biometrics	2	2
Cryptography issues in offline apps	1	5
Difficult feedback	2	3
Insecure infrastructure	5	13
Internet of Things	2	4
Issues of battery life	1	1
OS Permission Models	4	11
Physical access to device	2	7

Codes	Srcs	Refs
Privacy	1	1
Secure install channel	3	5
Upgrading issues	6	9
Architecture policies	4	6
Automation	1	1
Awareness of security	3	10
Deep learning	1	1
Augmented analysis	1	2
Augmented attacks	1	3
Augmented code review	2	3
Augmented test generation	2	2
Development team structure and working	5	11
Agile	2	4
Analyse system and changes	6	9
Interaction with other teams	3	10
Interaction with product owners	7	24
Keeping list of discovered exploits	1	1
Mistakes and errors	1	1
Right to query	3	4

Codes	Srcs	Refs
Standard development environments	1	4
Trade-off between security and cost	9	17
Dialectic	1	1
Automatic code review tools	4	8
Interaction amongst development team	3	10
Penetration and other testing	8	20
Reviews	5	15
Verification	1	1
Fundamental principles of software security	3	4
Openness	2	9
- Defensiveness	1	5
- Silos	2	6
- Time pressures	1	3
Co-location	1	5
Curiosity	1	2
Informal communication	1	2
Open source	6	10
Shared understanding	1	1
Social interaction	1	4

Codes	Srcs	Refs
Planning for future security	4	6
Secure by default	2	4
Security state machine	3	7
Step-by-step	4	5
Teaching and learning	2	4
Tick box security	8	27
- Complacency	2	3
- Discourages innovation	2	5
- Discourages updating	1	2
- Legalistic arguments	2	6
- Out of date	1	2
Google five privacy principles	1	1
Using cryptography	4	4
Whole system security	6	9
- Security as an add-on	1	2
Attacker profiling	6	17
Choice of tools environments components and protocols	10	17
Continuous upgrading	9	20
Exploit and threat analysis	10	16

Codes	Srcs	Refs
Humans as part of system	6	20
No absolute security	5	9
Prevent access	7	9
Requirements vs specification	2	3
Security budget	3	3