View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Lancaster E-Prints

I'd Like to Have an Argument, Please

Using Dialectic for Effective App Security

Charles Weir, Awais Rashid
Security Lancaster
Lancaster University
UK
{c.weir1, a.rashid}@lancaster.ac.uk

James Noble
ECS
Victoria University
Wellington, NZ
kjx@ecs.vuw.ac.nz

***Abstract*— The lack of good secure development practice for app developers threatens everyone who uses mobile software. Current practice emphasizes checklists of processes and security errors to avoid, and has not proved effective in the application development domain. Based on analysis of interviews with relevant security experts, we suggest that secure app development requires 'dialectic': challenging dialog with a range of counterparties, continued throughout the development cycle. By further studying the different dialectic techniques possible in programmers' communications, we shall be able to empower app developers to produce the secure software that we need.**

I. INTRODUCTION

There are now more than 2 billion smartphone users in the world. We use apps to communicate, apps to plan, apps to manage our finances, apps to do our shopping, and apps to remember our security credentials. Increasingly those apps are handling our sensitive personal information, and thus it is becoming vital to ensure our security and privacy.

Unfortunately, there is evidence that the developers of smartphone apps are not delivering this security. Enck et al. [19] used static analysis to study 1100 commercial Android apps in 2011, and found privacy issues in a majority of apps available to download. App security solution provider Bluebox analyzed the top five payment apps in 2015 [8] and found vulnerabilities permitting financial theft in all of them, along with further privacy issues. Both surveys suggested that the errors were avoidable; app programmers could have made choices that would have prevented the issues.

In addition, there is widespread concern about the problem. The Ponemon Institute carried out a IBM-funded survey in 2015 of 640 individuals from organizations developing apps in the US [41], and found that 77% believed that securing mobile apps was 'very hard', and that 73% percent believed that developer lack of understanding of security issues was a major contributor to the problem.

Permission to freely reproduce all or part of this paper for non-commercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
EuroUSEC '17, 29 April 2017, Paris, France
Copyright 2017 Internet Society, ISBN 1-891562-48-7
http://dx.doi.org/10.14722/eurousec.2017.23002

These studies demonstrate that existing industry practices are insufficient to provide the application security and privacy we need. To address this problem we could look for improvements to environments and APIs; we could look at tools to automate security improvements; or we can look at ways in which we can help app programmers themselves to improve security given existing constraints. All are valuable approaches; we chose the third option. The research question we formulated during the work is therefore:

What techniques and ideas lead to the development of better secure app software?

We found little existing research on what works well. Current practice emphasizes checklists of processes and errors, along with static analysis of code. To improve practice, however, we need an understanding of what works in the real world.

Since there is little existing theory on this subject, we had no basis for an experimental approach; we have no hypotheses to test. Written or email surveys could be useful to find out current practice from a list of options, but are unsuitable of the kinds of open question that generates theory. We therefore conducted a Constructivist Grounded Theory [11] study, involving face-to-face interviews with a dozen experts whose cumulative experience totaled well over 100 years of secure app development, to develop theory on the best techniques available for developers.

Our early analysis of the interviews [48,49] found a wide range of difference between interviewees, and concluded that the discipline is relatively immature. This paper builds on this earlier work and goes beyond by contributing a catalogue of techniques to empower app developers to deliver secure software. In identifying these techniques, we found surprising discrepancies between the current industry understanding of the approach required by app developers, and the experts' recommendations. Specifically we concluded that the management approach of process checklists offers little help to app developers; and that even 'whole system security' approaches do not get close enough to day-to-day programmer experience to be very useful.

Instead, we identified that the most important and successful secure development techniques share a quality we

call 'dialectic', meaning learning by challenging. These techniques use dialog with a range of counterparties to achieve security in an effective and economical way. The increase in security comes from the developers' continued interaction with the resulting challenges, not from passive learning.

This suggests that, like the Monty Python character who requests an argument [35], in the words in the paper's title, we need developers to actively seek out arguments and challenges.

The novel contribution of this work is to provide:

- A shift in perspective from artifacts to communication,
- An analysis of secure development practices that challenges conventional processes and checklists, and
- A theory linking a range of practices that encourage secure behaviors by app developers,

In the rest of this paper, section II explores existing work; section III explains the methodology in more detail; sections IV to X explore the techniques in detail, examining the problem each solves, and suggesting how teams typically implement each one; then section XI explores the techniques as a whole and suggests approaches for future research.

II. Related Work

We looked for related work in three areas:

- How programmers learn security,
- Resources to help programmers improve security, and
- Techniques to help teams improve

A. How Programmers Learn Security

We first consider work on how app developers learn about security. Balebako et al. surveyed and interviewed over 200 app developers, and concluded that most approached security issues using web search, or by consulting peers [5].

A survey by Acar et al. reached similar conclusions; and they also determined experimentally the surprising result that programmers using digital books achieved better security than those using web search [1]. Yskout et al. tested experimentally the effect of using security patterns in server design; the results suggested a benefit but were statistically inconclusive [52].

There are a number of papers exploring the reasons why programmers introduce weaknesses in mobile apps, especially Android, and possible reasons for them. For example, Egele et al. [17] studied the misuse of cryptographic APIs, concluding that it was widespread and required better APIs; Fahl et al. studied SSL use [20], concluded some apps were vulnerable to man-in-the-middle attacks, and suggested an improved API to solve the problems. However, this work has not been used to improve programmer performance.

B. Resources to Help Programmers

Turning to resources, one might expect that a very effective contribution to app security would be books explaining how to do secure app development. Unfortunately, few seem to exist covering a higher level than code. There are many good works describing the theory and practice of software security, such as Gollman's 'Computer Security' [24], Schneier's 'Secrets and Lies' [43] and Anderson's 'Security Engineering' [2]; these are particularly valuable for introducing the concepts of 'whole system security', but all work at a level that isn't helpful as anything but background reference for a software developer.

Instead, the most useful learning books for software developers do tend to be those that convey information in a relatively terse and readable form, and in manageable chunks. A good example is Howard et al.'s '24 Deadly Sins of Software Security' [27]; its format is similar to that of the patterns literature. Targeted specifically at particular platforms are books explaining the Android security model and development techniques; examples include 'Pro Android 4' [30] and 'Android Security Internals' [18]. For iOS there are equivalents, such as 'Learning iOS Security' [6]; though this is more a description of security features than a guide to avoiding security issues.

Some of the most popular[1] security books are the ones with a platform-specific 'Black Hat' (attacker) approach. For example the Android Hacker's Handbook [16], and its corresponding versions for iOS and web apps, contain a good deal about exploits against the operating system, a certain amount about analyzing existing apps, but little about how to guard against exploits as a developer. Chell's Mobile Hacker's Handbook [12] takes a similar approach, covering iOS, Android and even Blackberry platforms, and does provide limited advice for developers.

The community-written OWASP Top Ten Mobile Risks site [39] is a widely accessed[2] resource detailing specific programming issues and how to avoid them. Its authority and availability make it very effective, though again it does not consider 'whole system security' issues.

App programmers tend to use web search and discussion sites such as Stack Overflow as their primary source of information on security [1]. Unfortunately these lack overview discussions [7], making them 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 on security matters tend to be of questionable accuracy especially when they quote code. Acar et al. [1] analyzed answers on Stack Overflow to app security questions, worryingly finding around 50% of solutions to a set of security questions to contain insecure code snippets.

C. Techniques to Help Teams Improve

Two projects, by Xie et al. and Nguyen et al., [37,51] have developed IDE-based tools to teach programmers by detecting possible security flaws in Android developments. The approach is promising, but obviously requires programmers to adopt the tools; also no papers are yet available validating their effectiveness. Others, such as Near and Jackson, and Lerch et al., [31,36] have code analysis tools to detect security defects; these work but provide only limited feedback to developers.

One might expect the most effective approach to be a prescriptive set of instructions telling programmers what to do,

[1] Based on Amazon.com rankings as at January 2016

[2] Based on Google rankings in February 2016

a ‘Secure Software Development Lifecycle’ (SSDL) such as those promoted by Microsoft [34] and others. However Conradi and Dyba [14], among others, identified that programmers have difficulty with, and resist learning from, formal written routines, and this appears to have been the experience with SSDLs in practice [22]. So, since about 2010 several of the SSDLs have been replaced by ‘Security Capability Maturity Models’ [33,40], to allow management influence on software security at a corporate level based on measurements using checklists of processes used to improve security. These are effective [32] at defining what development teams should achieve, but provide little help to developers on how best to achieve it.

This means that for the majority of app developers who do not have the support of formal process-driven organizations [45] we need to find a lightweight, non-prescriptive approach.

D. Limitations of Existing Literature

Consistent in all this literature, is a lack of theory how to guide development teams to achieve security in the specification, architecture, and design activities. Instead, we see mainly checklists at for the coding activity alone.

Moreover, while ‘whole system security’ experts such as Anderson [2] 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.

III. Methodology

This section explains our choice of methodology, briefly outlines Grounded Theory, and introduces the research participants.

A. Choice of Methodology

Our purpose in the research was to generate knowledge about good approaches to secure development. Two perceptions drove the research approach:

- 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 we observe tend to be positive in outlook.

Our major resource was personal connections and links to industry specialists in app development, including in secure app development. Since our aim was to generate, rather than to test, theory, we chose Grounded Theory as our primary research method. Our study used semi-structured interviews over 6 months with a dozen such experts. To encourage positivity, we used elements of Appreciative Inquiry [15] in our questioning: the ‘Discovery’ of best practice and the ‘Dream’ of ideal practice.

B. Grounded Theory

Grounded Theory [23] uses textual analysis of unstructured text to make theory generation into a dependable process. It requires line-by-line analysis of everything relevant that is available to the researcher: interview transcripts, relevant research literature, field notes from observation and anything else that can reduce to text form. The process is iterative, with analysis of initial findings from interviews or similar leading to changes in the research thrust and direction.

We used the Constructivist GT variant [11], acknowledging the effects of the researcher on the results. We followed the principles for software engineering GT described by Stol et al. [44]. We recorded interviews and transcribed them manually; organization of the data used the commercial tool Nvivo; coding, memoing and sorting were all by the lead author.

The interviews consulted the participants as experts rather than as subjects. They addressed what each had found to be most successful in their experience in secure software development. Following the principles of Appreciative Inquiry, questions avoided discussion on what does not work and concentrated on the most effective practice known to each interviewee.

C. Research Participants

Though we chose the participants for their experience with app security, most drew on experience with a far wider range of software domains in answering the questions. Table 1 shows the interviewees, with an indication of organization size (S for solo or less than 10 people; M for up to 1000 people; and L for larger and government organizations), a suggestion of their years’ experience with app security, and a description of their main day-to-day role. All had worked extensively with UK or US-based organizations; all but P5, P8, and P10 had an original background as programmers.

Table 1: Interviewees, organization sizes, experience and roles

ID	S	Y	Typical Role
P1	S	2	Developing apps for business clients
P2	M	10	Leading large security-focused team
P3	L	8	Developing user-facing web services
P4	S	20	Designing and implementing smart card software
P5	M	5	Architecting and promoting a secure service
P6	L	20	Consulting on app and IoT security
P7	M	10	Developing and architecting OS services
P8	L	10	Architecting mobile phone operator services
P9	L	5	Designing and protecting web-based services
P10	S	15	Architecting and promoting app technologies
P11	L	10	Designing OS security enhancements
P12	S	20	Developing apps for business clients

In the following sections, quotations from the interviewees are in *italics*. We have edited them to protect confidentiality and indicate context: square brackets show additions and replacements; ellipses show removals.

IV. Introducing Dialectic

Grounded Theory emphasizes that the theory generated should cover the greatest variation in the data. Our initial analysis of the transcribed interviews [48] showed very significant variations between experts, suggesting that the discipline is relatively immature, and that our experts were

merely providing a taxonomy of all the 'whole system security' techniques.

Looking to explore in more detail how programmers were to achieve this taxonomy, we reanalyzed the interviews. We found to our surprise that they contained little mention of important parts of the 'Whole System Security' taxonomy: for example devising mitigations or using checklists of possible errors. Indeed some interviewees explicitly described these as unimportant.

Instead, we observed that the core theme was the nature of the developer's interaction with external parties: a friendly adversarial interaction. The word 'dialectic' had surfaced in our earlier work [48], and we realized it was the common theme to all these interactions. 'Dialectic' is the finding out of knowledge, especially logical inconsistencies, through one person questioning another. It first appeared as the technique used by the Greek philosopher Socrates in his dialogs.

Using dialectic addresses a particular weakness in much existing secure development research, namely that emphasis on code level security often misses possible exploits based on the functionality and system design – or may prove unnecessarily costly:

So implicit in [conventional thinking] is the notion that programmers decide what they are doing [only] in code … being told to put something in place without them understanding the greater implication. (P9)

Instead, developers need to think what approaches an attacker might use to gain benefit from the system they are producing, and then to decide how 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 a very different approach from 'normal programming'. It requires developers to think in different ways.

They are very devious. There are exploits that they have realized which are, well, you wouldn't really think like that if you were an engineer (P2)

It was not difficult to work out why our experts should view dialectic as a solution. Unlike other forms of software quality such as performance or reliability, security involves the idea of someone different: an attacker who will use very different ways of thinking. To deal with such threats, a developer needs to think 'outside the box'; the easiest way to achieve that is with challenges from others.

A. *Specific Dialectic Techniques*

From the GT analysis, we have drawn out six specific techniques that illustrate these kinds of dialectic challenges. These techniques are:

Brainstormed attacker profiles	Ideation sessions to derive possible attackers and attacks on the system
Negotiated security	Communicating security decisions in ways their stakeholders can understand, to prioritize 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.

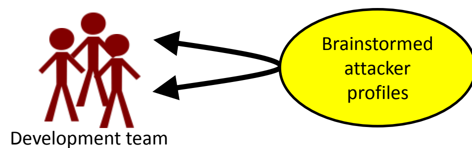
We observe that these techniques characterize themselves in terms of the source of the challenge to the programming team: other team members; tools; other roles in the software development process; and the consequences of end use. Sections V to X show the techniques, and the counterparties involved with each.

In each case, the dialectic continues throughout the development cycle; and in each case, it is always two-way: the experts state that the increase in security comes from the interaction with the challenges, not from a passive understanding of the challenge.

We do not have evidence to claim that dialectic provides the *best* techniques for achieving security; however, our study suggests that dialectic communication *strongly correlates with* effective software security.

Sections V to X explore each technique in detail. Each includes an illustration and illustrative quotation, followed by an 'Exploration' section discussing the context of the problem, and a 'Solution' section describing how the interviewees suggest addressing it. The academic paper format restricts the length of each description; fuller versions are available in the first author's MSc thesis [47]

V. BRAINSTORMED ATTACKER PROFILES



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 realized. (P2).

A. *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 realized that no system is ever unbreakable (P9)

As a result, secure development is not a matter of making a completely secure system. Instead, it becomes a question of which defenses to implement: 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)

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

B. Solution

Use brainstorming techniques to identify both attackers and possible exploits. Brainstorming is a form of dialectic that uses interaction with a range of people with different outlooks to create knowledge and ideas that were not obvious beforehand.

The first step generates 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 behavior. (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.

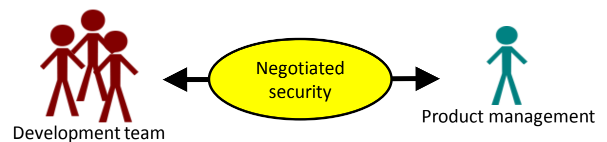
"One of the things I like to do with the [penetration testing] guys is to, if you sit down and say 'what are all the different ways you could subvert this system'. It is quite common to come up with 20, 30, 40, 50 in five or ten minutes of brainstorming. I bet you, you wouldn't think of half of them."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

An excellent concise recipe for running these sessions is in the seminal work on negotiation, Fisher et al.'s 'Getting to Yes' [21], whose chapter 'Invent Options for Mutual Gain' contains a step-by-step prescription for an effective brainstorming process. Although the illustration shows the brainstorming as being with others in the programming team, for solo developers it will typically be with clients, members of related technical teams, and other stakeholders.

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, and throughout the software product lifecycle.

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)

VI. NEGOTIATED SECURITY



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)

A. Exploration

Merely identifying the possible attackers and exploits does not itself deliver software security. The need is to prevent them from causing significant damage to users, stakeholders or others. So a development team takes the list of possible attacks, and works 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. How, though, do they make the decision which to implement?

Our interviewees conclude that 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.

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

B. Solution

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

[When I started] a project I'd go back and ask [my customer]... 'You do realize 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)

Here the dialectic is with the stakeholders. The stakeholders have knowledge of the importance of different kinds of security in the context of the development, that is likely to be very different from that of the developers, and it is important that this challenges the assumptions of the development team.

It is hard to over-emphasize the value of the interpretation skill. 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 security to implement 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. [38]. 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)

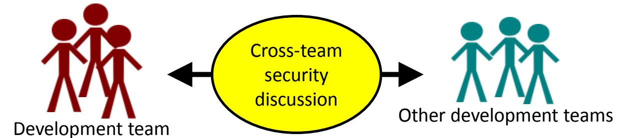
For simpler projects and systems, there may not be sufficient engagement from stakeholders to be able to do this kind of trade-off; in that case, it becomes the responsibility of the developer:

[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 to make these decisions; 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)

VII. Cross-Team Security Discussion



[What was very successful was] working incredibly closely as a team, and 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 don't want to expose their own domain. (P8)

A. 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 developers 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 organizational 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 … so the silos become self-reinforcing and it is very difficult sometimes to know whether you have actually been delivered all the facts. (P8)

Or where teams are effectively separated by time – they are not working on the project at the same time:

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

B. Solution

Ensure frequent and open communication on security problems in any way available. The effect of such communication is to challenge each team to address the security issues – another example of a dialectic process – and to allow casual communication about possible security problems. Ensure you have considered all the types of team: other programmers, operations, and security experts.

Bringing members of the different teams together on a social basis encourages the right 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, and 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 'falling between two stools' problems may happen, and is used, for example, in a secure development process introduced by the lead author [46]. Where multiple organizations are involved, the security scope 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)

VIII. 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)

A. Exploration

It is notoriously difficult to spot one's own errors. This is especially true when the errors are faults in complex reasoning, or are due to misunderstandings. Thus, 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 will always suffer to some extent from 'groupthink'; the need to generate a shared understanding brings with it the danger that that understanding may include misunderstandings and blind spots.

B. Solution

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

This is perhaps the most obvious example of dialectic in security – the counterpart queries the assumptions of the design and code, causing developers to review and change their understanding.

There are several common ways of arranging Security Challenge: pair programming, security review, code review, and penetration testing. Pair programming gives the developer the benefit of external questioning:

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 [42].

[We have] a separate security review system for, 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. (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 feedback any 'successful' exploits they have found to the development and operations teams.

[Ensuring software security] tends to get handed off, in most companies I've worked with, to a white-hat hacking team. [They] don't do it 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 for an app is an external security code review. Many companies now specialize 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)

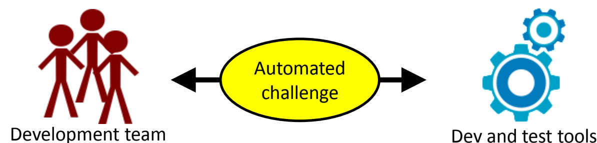
We do code reviews as much as possible. And I point out when I think something may have some issues, things like that. (P7)

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 [13]. However, the other three interventions all represent additional costs for an organization, 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)

Unfortunately, cost usually makes the external options, such as external code review or pen test, unsuitable for solo developers. An alternative is 'Rubber Duck Debugging' [28]: explaining thought processes to an anthropomorphized object.

IX. Automated Challenge



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

A. 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 organizations will not see value in paying for penetration testing or external reviewers, nor have skills to do either in-house.

Equally, it is a poor use of expensive resources to find problems that are cheap to find in other ways.

How do we achieve this?

B. Solution

Use software tools to create the dialectic. There are two areas where automation can help a great deal by challenging developers. 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 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… for anything using C or C++ is to 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 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 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 made a set of recommendations what is required [29]. The tools may be run infrequently, or as part of an automated build. Solo developers who lack a stable build system will typically run them as part of the release preparation process.

Automated security testing comes in two forms. First is the automation of tests that find security defects, 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 forms 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)

A second, recent, innovation is to use randomization and 'deep learning' techniques to enable tests that would not necessarily occur to a human tester:

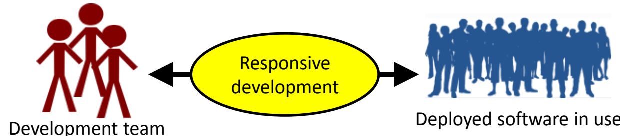
I actually find that our fuzzing efforts, which you could view as a form of code analysis, have quite a bit more tangible results. The fuzzing effort doesn't happen at code review time, but happens at check in time; we have clusters of machines where we are doing attacks against the software that is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

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.

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)

X. 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)

A. Exploration

To keep apps secure 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 the increasing 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 almost 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'; on completion of the initial app development phase, the development team is 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)

Moreover, it can be 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 behavior doesn't allow for the 'dipping into things'. (P8)

B. Solution

Instigate a long-term development approach to support both security monitoring and regular updating; include logging, and other feedback mechanisms within an app.

Here the dialectic challenges are from the external world – users, attackers, and any other influences. Since we need the dialectic to be both continuous and two-way, developers must use or create specific methods to get and monitor feedback from the apps, and project stakeholders need to ensure that projects have a continuous long-term support and monitoring element, with an explicit mechanism to deliver regular enhancements.

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 sanitize 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)

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)

A common solution, implemented in several cases by the lead author, for cases where the apps communicate with a server, is 'forced upgrades' based on the app version number. This requires extra support in the app and server: on startup 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 'store' app to make the upgrade.

Getting the resource to deliver regular enhancements requires a long-term approach to product development, since there will be costs long after the first release. Typically, organizations will 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 products that are currently in this three year window [are maintained] so not everything, but the current products, we will keep up to date. (P3)

Thus, we need development contracts ('maintenance contracts') and system architectures that allow for continued app development rather than the more traditional 'fire and forget' approach.

XI. Discussion

The ordering of the techniques is roughly chronological from the point of view of the development team. While developers use each repeatedly throughout the development cycle, they will encounter the need for Brainstormed Attacker Profiles and Negotiated Security earlier in each development cycle; and the interaction in Responsive Development naturally comes rather later.

A. Relationship to Existing Work

Comparing the existing work on app developers and security, section 2 identified valuable research on current practice by developers such as that by Balebako et al. [5]; this work goes further by identifying approaches for *better* practice. Much of the remaining literature we discussed is also valuable in the context the dialectic techniques as providing solutions to the challenges identified through dialectic. Thus an Android app programmer who is made aware of security issues from Brainstormed Attacker Profiles, Security Challenge or Automated Challenge would then be motivated to search for solutions on the web or in practitioners' literature such as 'Android Security Internals' [18]; work by Acar et al. [1] suggests that their best choice would be the book .

Considering the dialectic 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. For Automated Challenge, there is a considerable range of automated validation tools available even if, as found by Johnson et al. [29], these are currently not often used by developers.

The other three techniques, Brainstormed Attacker Profiles, Negotiated Security, and Cross-Team Security Discussion are less well reflected in existing security literature; section D proposes approaches to research them. However, they do have parallels in other aspects of software engineering. Brainstormed Attacker Profiles relates to the HCI concept of 'personas'; Atzeni et al. [4], describe an analytic (rather than brainstorm) approach to generating attacker personas. Cross-team Security Discussion relates to the large amount of work available on collaboration between distributed teams [10]. Related to Cross-team Security Discussion, some new work by Ashenden and Lawrence [3] uses an Action Research approach to improve the effectiveness of the relationships between security professionals and software developers.

B. Improvements on Existing Practice

Section II.D identified that existing practice specifies little about the team interactions required to achieve software security. By contrast, the dialectic techniques 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 a new way of looking at communication.

For developers, dialectic provides a set of attitudes to development teamwork and approach. It meshes effectively with agile self-organizing teams [26]; the developers themselves can choose what processes and deliverables best serve their 'dialectic'. What is important is the difference in the ways developers interact with the world.

C. Research Validity

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

Considering first Conclusion Validity, do the research data justify the conclusions? Grounded Theory's rigorous process of line-by-line coding, categorization, and sorting generates a 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 dialectic 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 [25] 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.

In terms of External Validity, can the results be generalized to a wider scope? GT's conclusions are always limited to the specific scope studied [11]. 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.

D. Future Work

Considering the dialectic techniques themselves, we suggest that three are reasonably well understood and researched in various ways: Security Challenge, Automated Challenge, and Responsive Development. We propose further research to examine the three less well-understood techniques: Brainstormed Attacker Profiles, Negotiated Security, and Cross-team Security Discussion.

It would ask research questions along the lines of:

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

- How best do we represent security questions in business terms?
- What forms of cross-team interaction are most effective to ensure app security?
- Can team members learn the dialectic approach, or is it best used to inform the management and structure of teams?

There are several possible approaches to this research. Experimental approaches might 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. Alternatively, a survey approach might ask the questions of a variety of developers and stakeholders to produce a possible consensus.

A different and potentially very rewarding area of research is in further dialectic techniques. For example De Bono [9] has defined a variety of 'Lateral Thinking' techniques to help teams and individuals to challenge their thinking.

A third area of research is in techniques to introduce the dialectic approach to developers. A majority of app developers work independently or in relatively small organizations [45]. The authors have proposed ways to introduce effective security techniques to such individual developers in a recent paper [50]. These include:

- Educational app games for developers to play,
- Storytelling, through blogs or even TV storylines, and
- Massively online courses, and TED-style video

We see these approaches as providing a path to help the dialectic techniques to have an impact on development practice in the next couple of years.

XII. Conclusion

In a rigorous study using interviews of experts in secure app development, we found three novel aspects. First was an emphasis on programmers' activities, in addition to the artifacts delivered. Second was a discrepancy between current industry understanding of good security practice, and experts' recommendations. Last was a theory that emphasises the nature of a specific kind of communication and interaction with developers: dialectic.

We concluded that techniques for software security may best be expressed in terms of the dialectic nature of the developers' own interactions, and not solely in terms of formal processes, artefacts and reports.

Section 4 describes six dialectic techniques, each involving continuous challenging dialog with a different counterparty. We suggest that these techniques are well suited for app development teams in the majority of organizations. We shall investigate the techniques further as discussed in section XI.D; we shall also look for ways to disseminate them more widely, and research interventions to introduce them into a range of existing development teams.

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

XIII. References

[1] 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.

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

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

[4] Atzeni, A., Cameroni, C., Faily, S., Lyle, J., and Flechais, I. Here's Johnny: A Methodology for Developing Attacker Personas. *2011 Sixth International Conference on Availability, Reliability and Security*, IEEE (2011), 722–727.

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

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

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

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

[9] De Bono, E. *Lateral Thinking : Creativity Step by Step*. Harper & Row, 1970.

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

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

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

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

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking SSL Development in an Appified World. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*, (2013), 49–60.

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

[22] Geer, D. Are Companies Actually Using Secure Development Life Cycles? *IEEE Computer June*, 2010, 12–16.

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

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

[25] 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.

[26] 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.

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

[28] Hunt, A. and Thomas, D. *The Pragmatic Programmer : From Journeyman to Master*. Addison-Wesley, 2000.

[29] 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.

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

[31] 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.

[32] McGraw, G. Four Software Security Findings. *Computer 49*, 1 (2016), 84–87.

[33] McGraw, G., Migues, S., and West, J. Building Security In Maturity Model (BSIMM7). 2016. https://go.bsimm.com/hubfs/BSIMM/BSIMM7.pdf.

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

[35] Monty Python. The Argument Sketch. http://www.montypython.net/scripts/argument.php.

[36] 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.

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

[38] 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.

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

[40] OWASP. Software Assurance Maturity Model Project. https://www.owasp.org/index.php/OWASP_SAMM_Project.

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

[42] 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.

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

[44] 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.

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

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

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

[48] Weir, C., Rashid, A., and Noble, J. How to Improve the Security Skills of Mobile App Developers: Comparing and Contrasting Expert Views. *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016): Workshop on Security Information Workers*, USENIX Association (2016).

[49] Weir, C., Rashid, A., and Noble, J. Early Report: How to Improve Programmers' Expertise at App Security? *1st International Workshop on Innovations in Mobile Privacy and Security Co-Located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2016)*, CEUR-WS.org (2016), 49–50.

[50] 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).

[51] 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.

[52] 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.