Report from Dagstuhl Seminar 19231

Empirical Evaluation of Secure Development Processes

Edited by

Adam Shostack[1], Matthew Smith[2], Sam Weber[3], and Mary Ellen Zurko[4]

1 **Seattle, US,** adam@shostack.org
2 **Universität Bonn and Fraunhofer FKIE, DE,** smith@cs.uni-bonn.de
3 **Carnegie Mellon University – Pittsburgh, US,** smweber@andrew.cmu.edu
4 **MIT Lincoln Laboratory – Lexington, US,** mez@alum.mit.edu

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 19231 "Empirical Evaluation of Secure Development Processes". It includes a discussion of the motivation and overall seminar organization, the abstracts of the talks and a report of each working group.

Seminar June 2–7, 2019 – http://www.dagstuhl.de/19231
2012 ACM Subject Classification Security and privacy → Software security engineering, Human-centered computing, Software and its engineering → Software development process management, Software and its engineering → Designing software, Security and privacy → Systems security
Keywords and phrases security, security engineering, secure development lifecycle, empirical software engineering, empirical security engineering
Digital Object Identifier 10.4230/DagRep.9.6.1
Edited in cooperation with Daniel Votipka

1 Executive Summary

Adam Shostack
Matthew Smith
Sam Weber
Mary Ellen Zurko

The problem of how to design and build secure systems has been long-standing. For example, as early as 1978 Bisbey and Hollingworth[6] complained that there was no method of determining what an appropriate level of security for a system actually was. In the early years various design principles, architectures and methodologies were proposed: in 1972 Anderson[5] described the "reference monitor" concept, in 1974 Saltzer[7] described the "Principle of least privilege", and in 1985 the US Department of Defense issued the Trusted Computer System Evaluation Criteria[8].

Since then, although much progress has been made in software engineering, cybersecurity and industrial practices, much of the fundamental scientific foundations have not been addressed – there is little empirical data to quantify the effects that these principles, architectures and methodologies have on the resulting systems.

This situation leaves developers and industry in a rather undesirable situation. The lack of this data makes it difficult for organizations to effectively choose practices that will cost-effectively reduce security vulnerabilities in a given system and help development teams achieve their security objectives. There has been much work creating security development lifecycles, such as the Building Security In Maturity Model[1], Microsoft Security Development LifeCycle[3] OWASP[4] and ISECOM[2] and these incorporate a long series of recommended practices on requirements analysis, architectural threat analysis, and hostile code review. It is agreed that these efforts are, in fact, beneficial. However, without answers as to why they are beneficial, and how much, it is extremely difficult for organizations to rationally improve these processes, or to evaluate the cost-effectiveness of any specific technique.

The ultimate goal of this seminar was to create a community for empirical science in software engineering for secure systems. This is particularly important in this nascent of research in this domain stage since there is no venue in which researchers meet and exchange. Currently single pieces of work are published at a wide variety of venues such as IEEE S&P, IEEE EuroS&P, ACM CCS, USENIX Security, SOUPS, SIGCHI, ICSE, USEC, EuroUSEC, and many more. The idea was that bringing together all researchers working separately and creating an active exchange will greatly benefit the community.

Naturally, community-building is a long-term activity – we can initiate it at a Dagstuhl seminar, but it will require continuous activity. Our more immediate goals were to develop a manifesto for the community elucidating the need for research in this area, and to provide *actionable* and *concrete* guidance on how to overcome the obstacles that have hindered progress.

One aspect of this was information gathering on how to conduct academic research which is able to be transitioned and consumed by developers. We felt that all too frequently developer needs aren't fully understood by academics, and that developers underestimate the relevance of academic results. Our information gathering will help foster mutual understanding between these two groups and we specifically looked for ways to build bridges between them.

A second obstacle which we aimed to address is how to produce sufficiently convincing empirical research at a foundational level as well as in the specific application areas. Currently there is no consensus on what are ecologically valid studies and there are sporadic debates on the merits of the different approaches. This seminar included a direct and focused exchange of experience and facilitated the creation of much needed guidelines for researchers. In accordance with our bridge building, we also looked at what developers find convincing, and how that aligns with research requirements.

Seminar Format

Our seminar brought together thirty-three participants from industry, government and both the security and software engineering academic communities. Before the seminar started we provided participants with the opportunity to share background readings amongst themselves.

We began our seminar with level-setting and foundational talks from industrial, software engineering and security participants aimed to foster a common level of understanding of the differing perspectives of the various communities.

Following this the seminar was very dynamic: during each session we broke into break-out groups whose topics were dynamically generated by the participants. The general mandate for each group was to tackle an aspect of the general problem and be actionable and concrete: we wished to avoid vague discussions of the difficulties involved with studying secure development but instead focus on how to improve our understanding and knowledge. After each session we met again as a group and summarized each group's progress.

At the conclusion of the seminar we brought together all the participants in a general discussion about further activities. In all, a total of eighteen further activities, ranging from papers to research guideline documents, were proposed and organized by the participants.

References

1 Building security in maturity model. http://www.bsimm.com/.

2 Isecom. http://www.isecom.org.

3 Microsoft security development lifecycle. https://www.microsoft.com/en-us/securityengineering/sdl/.

4 Owasp. https://www.owasp.org.

5 Anderson, J. P. Computer Security Technology Planning Study, Volume II. Tech. Rep. ESD-TR-73-51, 1972.

6 Bisbey, R., and Hollingworth, D. Protection analysis: Final report. *Information Sciences Institute, University of Southern California: Marina Del Rey, CA, USA, Technical Report ISI/SR-78-13* (1978).

7 Saltzer, J. Protection and the control of information sharing in Multics. *Communications of the ACM 17*, 7 (1974), 388–402.

8 United States Department of Defense. Trusted computer system evaluation criteria (orange book).

2 Table of Contents

3 Overview of Talks

3.1 Experience with the Microsoft Security Development Lifecycle

Steven B. Lipner (SAFECode – Seattle, US)

Main reference Michael Howard, Steve Lipner: "The Security Development Lifecycle", Microsoft Press, 2006
URL https://blogs.msdn.microsoft.com/microsoft_press/2016/04/19/free-ebook-the-security-development-lifecycle/

Between 2002 and 2004, Microsoft made a major commitment to software security, first executing a series of "security pushes" on major products and then introducing the Security Development Lifecycle (SDL), a mandatory process for improving products' resistance to attack. After the introduction of the SDL in 2004, the process requirements were updated periodically in response to new classes of attacks and new techniques for improving product security. This presentation summarizes the philosophy and practicalities underlying the SDL and the ways it has evolved, and outlines some of the ways that the effectiveness of the SDL and similar processes can be measured.

3.2 Security in Modern Software Development

Olgierd Pieczul (IBM – Dublin, IE)

Recent shifts in software engineering make the traditional view of software security obsolete. Secure development practices rely on several aspects of software engineering such as development process, tools and languages, developer skillset and work environment which all are rapidly changing. We observe security practices lagging behind and slowing down the transformation while increasing cost and reducing overall security.

Today's software delivery cycle can be as short as days or even hours. However, the security quality and compliance processes, reviews, testing and sign offs have been designed for long waterfall-style cycles. Similarly, while applications are being developed as small, independent microservices, security practices operate at a scale of system as a whole. These tasks tend to be labor intensive, managed, executed and reviewed manually and long to perform. This results in slowing down the delivery and reducing security assurance to rudimentary, checkbox-style level only.

This presentation covers the aspects of modern software development that have major impacts on security. We also identify and describe key areas of development that need to be considered, or much more significant considered, by security research, in particular: development process, software stack, team structure, developer skills and training and code reuse.

3.3 Software Engineers are People Too: Applying Human Centered Approaches to Improve Software Development for Security

Brad A. Myers (Carnegie Mellon University – Pittsburgh, US)

Main reference Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, YoungSeok Yoon: "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools", IEEE Computer, Vol. 49(7), pp. 44–52, 2016.
URL http://dx.doi.org/10.1109/MC.2016.200

Software engineers might think that human-computer interaction (HCI) is all about improving the interfaces for their target users through user studies. However, software engineers are people too, and they use a wide variety of technologies, from programming languages to search engines to integrated development environments (IDEs). And the field of HCI has developed a wide variety of human-centered methods, beyond lab user studies, which have been proven effective for answering many different kinds of questions. In this talk, I will use examples from my own and other's research relevant to security to show how HCI methods can be successfully used to improve the technologies used in the software development process. For example, "Contextual Inquiry" (CI) is a field study method that identifies actual issues encountered during work, which can guide research and development of tools that will address real problems. We have used CIs to identify nearly 100 different questions that developers report they find difficult to answer, which inspired novel tools for reverse-engineering unfamiliar code and for debugging. We used the HCI techniques of Paper Prototyping and Iterative Usability Evaluations to improve our programming tools. Through the techniques of Formal User Studies, we have validated our designs, and quantified the potential improvements. Current work is directed at improving the usability of APIs, using user-centered methods to create a more secure Blockchain programming language, addressing the needs of data analysts who do exploratory programming, helping programmers organize information found on the web, and helping end-user programmers augment what intelligent agents can do on smartphones.

References

1 Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. *Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools*. IEEE Computer, Special issue on UI Design, 49, issue 7, July, 2016, pp. 44-52

3.4 A Series of Experiments on Software Design Patterns

Walter Tichy (KIT – Karlsruher Institut für Technologie, DE))

Main reference Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, Lawrence G. Votta: "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions", IEEE Trans. Software Eng., Vol. 27(12), pp. 1134–1144, 2001.
URL https://doi.org/10.1109/32.988711

Software Design Patterns are proven solutions for software design problems. They are claimed to improve software quality, programmer productivity, and communication among developers, among others. A series of three experiments tests these claims.

The first experiment checks whether the presence or absence of design pattern documentation makes a difference [1]. Subjects received programs which contained design patterns and were asked to perform maintenance tasks on them. The experiment group received a few lines (20-30) of extra documentation pointing out the design patterns. Results indicate that documenting design patterns speeds up maintenance tasks that involve those patterns, or reduces defects. This experiment has been repeated using a different programming language with consistent results.

The second experiment compares maintenance tasks on programs with and without design patterns [2]. Four programs were implemented in two versions each: One with patterns and one in a modular fashion without design patterns. The results show that not all patterns are equally effective. Some speed up pattern-relevant maintenance tasks, some have no effect, and some require extra time. The complexity of the patterns seems to play an important role. This experiment has been replicated yielding similar results.

The third experiment tests whether programmers communicate more effectively with shared knowledge of design patterns [3]. Pairs of programmers were audio and video recorded when discussing maintenance tasks. The recordings were analyzed and the contributions of each participant were counted. The results clearly show that with shared pattern knowledge, a more balanced communication results, where balanced means that team members contribute equally to the discussion.

These three experiments (and their replications) support and complement each other and confirm the hypothesized positive effects of design patterns.

(The experimental techniques employed might also be of interest. Double counterbalancing helps neutralize sequencing effects. Communication lines as a measure of effective communication might be interesting in other contexts.)

References

1 Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. *Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance*. IEEE Trans. On Software Engineering, 28(6), June 2002, 595-606.

2 Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler and Lawrence G. Votta. *A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions*. IEEE Transactions on Software Engineering, 27(12), 1134-1144, 2001.

3 Barbara Unger, Walter F. Tichy. *Do Design Patterns Improve Communication? An Experiment with Pair Design*. May 2000, in WESS 2000. http://ps.ipd.kit.edu/176_766.php.

3.5 Empiricism in Software Engineering and Secure Systems

Laurie Williams (North Carolina State University – Raleigh, US)

The science of software engineering and the science of security can be advanced through the use of sound research methodologies while conducting empirical studies. With the use of sound methodologies, the evidence produced by a study is more credible, convincing, and substantiated and can be built upon by future researchers. As a result, the research results have more impact on other researchers and on practitioners. Additionally, meta-analysis and theory/law building is enabled when the research results are thoroughly reported.

Over the last twenty or more years, empirical software engineering researchers have conducted explicit efforts to mature the use of sound methodologies by creating guidelines and examples that bring established research practices into the context of software engineering. These guidelines include books and journal papers. Additionally, communities have come together with the goal of advancing software engineering research methods, including Dagstuhl seminars (Seminars 06262 and 10122), the International Software Engineering Research Network (ISERN), and the Empirical Software Engineering and Measurement (ESEM) conference. Over time, the top software engineering conference began to insist on the use of sound research methods which raised the bar for the community. A comparison of the research and validation methods in accepted papers in the 2002 International Conference on Software Engineering (ICSE) [1] and the 2016 ICSE demonstrated that the 2016 papers [2, 3] were more likely (3% versus 19%) to be an empirical reports which were more likely (0% versus 30%) to be accepted when compared with the 2002 papers. Additionally, the 2016 papers were more likely to have a formal evaluation (5% versus 35%) while papers that used example as their evaluation or to have no evaluation (20% and 7%, respectively, in 2002) were essentially non-existent in 2016.

Empiricism in emerging in security research. To establish a baseline, Carver et al. [4, 5] analyzed the evidence of science papers in the top security conferences, ACM Computer and Communications Security (CCS) and IEEE Security and Privacy. Their main motivation was to assess whether the papers reported information necessary for three key pillars of scientific research: replication, meta-analysis, and theory building. They examined the papers for completeness such that other researches would be enabled to understand, replicate, and build upon the results but looking for: research objectives, subject/case selection, the description of the data collection procedures, the description of the data analysis procedure, and the threats to validity. They found that 80% of the papers did not provide clearly-defined and labeled research objectives to define the goals, questions, and/or hypotheses of the research. Additionally, 70% had no discussion of the threats to validity or limitations of the work such that future researchers can make design choices to address these threats and limitations and to contextualize meta-analysis. These results indicate a need for the security research community to go through the type of maturation in empirical research that the software engineering community has gone through, including publishing guidelines in books and journal and the establishment of a community to drive this maturation.

References

1 M. Shaw, *Writing Good Software Engineering Research Papers* Proceedings of 25th International Conference on Software Engineering (ICSE'03), pp. 726-736, 2003.

2 C. Theisen, M. Dunaiski, L. Williams and W. Visser, *Writing Good Software Engineering Research Papers: Revisited* 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, 2017, pp. 402-402.

3 C. Theisen, M. Dunaiski, L. Williams and W. Visser, *Software Engineering Research at the International Conference on Software Engineering in 2016* SIGSOFT Softw. Eng. Notes, Vol. 42, No. 4, pp. 1-7. Buenos Aires, 2017, pp. 402-402.

4 Morgan Burcham, Mahran Al-Zyoud, Jeffrey C. Carver, Mohammed Alsaleh, Hongying Du, Fida Gilani, Jun Jiang, Akond Rahman, özgür Kafalı, Ehab Al-Shaer, and Laurie Williams. *Characterizing scientific reporting in security literature: An analysis of ACM CCS and IEEE S&P papers* In Proceedings of the Hot Topics in Science of Security: Symposium and Bootcamp, HoTSoS, pages 13–23, New York, NY, USA, 2017. ACM.

5 Jeffrey C. Carver, Morgan Burcham, Sedef Akinli Kocak, Ayse Bener, Michael Felderer, Matthias Gander, Jason King, Jouni Markkula, Markku Oivo, Clemens Sauerwein, and

Laurie Williams. *Establishing a baseline for measuring advancement in the science of security: An analysis of the 2015 IEEE Security & Privacy proceedings* In Proceedings of the Symposium and Bootcamp on the Science of Security, HotSos '16, pages 38–51, New York, NY, USA, 2016. ACM.

3.6 "Usable Security" approaches to Empiricism for Secure Software Development

Matthew Smith (Universität Bonn and Fraunhofer FKIE, DE), Sascha Fahl (Leibniz Universität Hannover, DE), and Michelle L. Mazurek (University of Maryland – College Park, US)

Usability problems are a major cause of many of today's IT-security incidents. Security systems are often too complicated, time-consuming, and error prone. For more than a decade researchers in the domain of usable security (USEC) have attempted to combat these problems by conducting interdisciplinary research focusing on the root causes of the problems and on the creation of usable security mechanisms. While major improvements have been made, to date USEC research has focused almost entirely on the non-expert end-user. However, many of the most catastrophic security incidents were not caused by end-users, but by developers. Heartbleed and Shellshock were both caused by single developers yet had global consequences. Fundamentally, every software vulnerability and misconfigured system is caused by developers making mistakes, but very little research has been done into the underlying causalities and possible mitigation strategies. In this talk we will explore the need for empiricism for secure software development in several application areas, including TLS, passwords, malware analysis and vulnerability analysis.

3.7 How the usable security community does developer studies: Field-ish studies with Build It, Break It, Fix It

Michelle L. Mazurek (University of Maryland – College Park, US)

Joint work of Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, Piotr Mardziel
Main reference Andrew Ruef, Michael W. Hicks, James Parker, Dave Levin, Michelle L. Mazurek, Piotr Mardziel: "Build It, Break It, Fix It: Contesting Secure Development", in Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pp. 690–703, ACM, 2016.
URL http://dx.doi.org/10.1145/2976749.2978382

Lab studies, field measurements, and field studies all contribute valuable knowledge to our understanding of secure development, but they also all have important drawbacks in terms of internal and external validity. The Build It, Break It, Fix It competition represents a new point in this tradeoff space. Teams compete over several weeks to build software according to a spec, gaining points for functionality and performance, then compete to break each others' software, causing the vulnerable team to lose points. The competition provides insight into how and why certain vulnerabilities arise, providing more control than a field study but more ecological validity than a smaller-scope lab study.

4 Working groups

4.1 Building Code Breakout

Adam Shostack and Carl Landwher

Joint work of Adam Shostack, Carl Landwehr, Florian Alt, Michael Coblenz, Serge Egelman, Shamal Faily, Joseph Hallett, Trent Jaeger, Mike Lake, Steve Lipner, Fabio Massacci, Simon Ou, Olgierd Pieczul, Riccardo Scandariato, Laurens Sion, Laurie Williams, Sam Weber

Building codes for physical structures have been developed over centuries to address societies' needs for buildings that are safe for their occupants and can stand up to customary natural threats – rain, wind, fire, earthquakes – for the areas in which they are built. Although such codes are diverse, they generally call for an initial approval by a local authority of the design and specification of the proposed structure, use of approved materials and methods in its construction, inspections to be carried out by trained third-party inspectors during construction, and a final approval for occupancy before the building can be occupied.

The construction and deployment of software systems, with few exceptions, is not subject to these kinds of third-party inspections and approvals. It has been proposed that the kinds of mechanisms found in the building code model might help achieve better security in the software systems deployed in, for example, medical devices, power grids, and "Internet-of-Things" devices.

The group considered what limits the building code metaphor might have in relation to software systems, what hypotheses about this approach might be developed and subjected to empirical evaluation, and what a roadmap might look like that would lead to the appropriate development and use of building codes for software systems with security and safety responsibilities.

4.1.1 Limits to the Metaphor

The discussion revealed many ways in which building codes for physical structures might reasonably be applied to software structures with security and safety responsibilities. The blueprint for a building specifies a structural design and plan for implementation and must be approved by a local authority before construction can proceed. Documents capturing system and software requirements and design have a similar role in software construction, though often implementation may proceed based on knowingly incomplete specifications and designs. Changes in the software requirements and design as construction proceeds are common in software, but they also occur in building construction.

Building codes normally require sequenced inspections of aspects of the building during construction: footings and foundations must be inspected before floors and walls are raised; plumbing and heating systems must be inspected before the walls are closed up and this work is hidden, and so on. Finally, when construction is complete a final inspection is required before the local authority certifies that the building can be occupied. As software systems are built there are often code reviews, unit and subsystem tests, and finally system testing before the software is released. In some environments, an authority may required to issue a final approval to operate before the system is made available to its users.

The metaphor was found to be limited in that software has no physical manifestation beyond the bits representing the programs, software can be replicated nearly without cost, and it can be transported easily around the globe. Buildings of course have substantial mass and are not easy to alter, move, or replicate. Physical structures change, but much more slowly than software, much of which is subject to continual updates. Inspectors of physical structures require some training typically measured in weeks or months; inspectors of software generally require years of training.

The group's conclusion was that despite the limitations, the building code metaphor seems to apply well to software with security responsibilities.

4.1.2 Hypotheses to Test / Questions to be Answered

The discussion on generating hypotheses to test opened by considering some characteristics desired of a proposed building code. As a principle, the building code should result in secure code and not in evidence to be consumed by the code inspector. Whatever evidence has to be tied to the code and the close the better. It was observed that the Common Criteria scheme has led to development of required documentation after the fact. These documents are then reviewed without reference to the actual underlying software, resulting in relatively low assurance of security at relatively high cost. Future building codes for software should avoid this trap.

The Windows Logo program, which enables developers of Windows applications to display the Windows logo only if their software passes certain automated tests, and the Apple Store program, which only accepts applications that pass both automated tests and some human review, were discussed and found to be within the category of software building codes.

In the context of an imagined building code for software with security responsibilities:

1. Would an inspector be required to exercise judgement, or could inspection be effectively automated?
2. What type of specification would be sufficient as the basis for effective enforcement (e.g., would a set of use cases or "user stories" be adequate? Would a data flow diagram specification suffice?
3. Does increasing the number of acceptance criteria incentivize better designs?
4. Will implementing an SDL reduce errors in implementations?
5. Will having a developing and applying a building code be better than doing nothing at all?

4.1.3 Working Notes and Ideas to Develop

1. Examples – Are the Apple store and Windows Logo "building codes"? What about the UK's DMCS IoT standard?
2. Principle: The building code should result in secure code and not in evidence to be consumed by the code inspector. Whatever evidence has to be tied to the code and activities intended to deliver products, and the closer the better. (Contrast: work done solely to satisfy the building inspector.)
3. Experiments on the building code: at every iteration a product developed to meet the code should be distinguishable from a product that has not been produced accordingly. Eventually we should be able to demonstrate that software systems built to the code were not just distinguishable, but better than without a building code.
4. Introduction of new technology for which verification is not applicable could be tested in unregulated area first and then moved to regulated areas when tools becomes available. Also, what happens when "the laws of physics" change?
5. Scalability of "testin" wrt the cost of the testing (e.g. CE label). Do we know if the types of "testing" experiments that we can run are linear or with exponentially decreasing returns? So that if you achieve a threshold that would enough because the next level

would cost you exponentially more (in either time or costs) while only delivering minor increasing quality.

6. Domain and time applicability Should a code commit to a building code version at production time and stick to it or must comply with the latest version at delivery time? What about "capricious" rules? What about late-breaking changes to the laws of physics? Building codes don't need to deal with those.
   We should distinguish between the "experiments" on the final product and "experiments" on the production process

4.1.4 Building Code Components

Building codes vary widely around the world, but in this subsection we describe a typical code for physical structures, which has aspects that occur at multiple phases of construction.

Before a building is even planned, separate goals are defined for structural integrity, fire safety, plumbing, drainage, electrical, HVAC and ADA compliance. Building codes constrain both a new building's specification and its design. After a building is designed, permitting is done with the municipality with paper copies of the documents being filed and sometimes a physical examination of the site is done. At each phase of physical construction (foundation laying, framing, etc) on-site inspections are done. When completed, a certificate of occupancy needs to be obtained, which also involves a separate inspection. Additionally, in order to maintain compliance with the building code, additional inspections are required over time to ensure that elevators, major appliances and such-like continue to be safe.

4.2 Ecological validity and study design for empirical secure development studies

Michelle L. Mazurek (University of Maryland – College Park, US) and Daniel Votipka (University of Maryland – College Park, US)

Joint work of Michelle L. Mazurek, Daniel Votipka, Tamara Denning, Heather Richter Lipford, Adam Aviv, Tobias Fiebig, Mathew Smith, Brendan Murphy, Michael Coblenz, Eric Bodden, Sascha Fahl, Mary Ellen Zurko, Walter F. Tichy, Walter, Brad Myers, Charles Weir

The design of a study is critical to producing correct results and the secure development researcher faces several unique and difficult challenges when attempting to choose the right design. For example, most development tasks are performed over weeks of collaborative work, making it difficult to even simulate ecologically valid conditions in a controlled lab setting. Also, security is typically a secondary focus in real-life settings, requiring further complication of the study design to avoid unrealistic behaviors. Further, measuring *correct* behaviors with respect to security is difficult as it generally requires proving the *absence* of any errors. Developer experience can also vary widely across many dimensions including programming ability, development workflow and security knowledge, reducing the generalizability of results without a broad sample of participants.

Fields such as software engineering [4] and usable security [5] offer recommendations for study design to resolve some of these issues. Other recent work has focused on specific challenges of secure development measurement, i.e., whether remote participants produce similar results as local participants; on the differences between using students, GitHub users, freelancers, and corporate developers [1, 2, 3]; and the effect of varying incentive structures [3].

Unfortunately, little general guidance specific to secure development study design currently exists for newer researchers, and several specific questions still remain open. Therefore, there is a need both for outlining best practices for study design based on work in other fields, and also investigating open questions specific to secure development tasks. As a first step toward addressing these issues, this working group focused on three specific tasks:

- Define a set of generic guidelines for new researchers and highlight security-specific examples of their application.
- Identify methodological issues requiring further research
- Identify a set of possible metrics for measuring aspects of secure development

4.2.1 Design Guidelines

The first group identified an initial set of generic guidelines useful to empirical research broadly, along with secure-development-specific examples for each. We list these guidelines and examples here.

1. Leverage well-established study-design practices from other fields

   Many of the decisions faced when designing a empirical evaluation of secure development are not unique to this domain of research. Other fields such as software engineering, human-computer interaction, psychology, sociology, and anthropology have established best practices that address many of these issues. For example, the *Guide to Advanced Empirical Software Engineering* offers best practices for topics such as survey design, statistical analysis, and research ethics. Researchers should consider these best practices for decisions that are not specific to secure development research.

2. Use pilot studies to hone in on the research objective and be willing to iterate on the study design

   The initial study design is rarely perfect. The research team typically does not have the necessary domain knowledge to predict how participants will interpret survey questions, task descriptions or interface element or how they will respond during the study. Therefore, it's important that all studies begin with a pilot where participants can share their impressions of the study to ensure the design actually meets the stated research objectives or whether new objectives should be sought. Similarly, as the study is being carried out, the researchers should monitor data collected and be open to making updates to the design if it becomes evident that changes are necessary.

3. Consider the metrics appropriate for the study methodology and desired outcomes

   One of the most important considerations when designing any study is in determining what data to collect. This includes what data to collect about the participant themselves and study-specific data, e.g., interactions with a tool or survey responses. Significant care should be taken to ensure that both the variables under study along with any potentially confounding effects are measured as precisely as possible. This is particularly difficult in security where *correct* development solutions require the absence of errors. Therefore, the researcher must consider designs that limit the scope of possible *correct* solutions, making analysis more manageable.

4. Consider the level of expertise of participants

   Because developers can vary drastically with respect to their secure-development expertise, it is important that the research team consider whether and how these variations will affect the study. For example, when A/B testing a tool to support vulnerability discovery, the researcher should consider the participants' security expertise. Because developers with more expertise are more likely to find a vulnerability irrespective of the tool being tested, their expertise should be measured and considered as a covariate during analysis. Alternatively, the participants could be provided with additional training at the start of the experiment to ensure all participants begin on equal footing.

5. Create a context for the experiment that appropriately prompts participants

   Participants' behaviors are typically context-dependent. This is especially true for security and privacy decisions [6]. Thus, the context in which participants are surveyed or observed should be selected to best fit the experiment's goals. For example, a researcher whose goal is to measure the impact of a new penetration testing tool at improving vulnerability discovery could run the study as part of a Capture-the-Flag exercise, where participants are instructed to exploit a series of programs, encouraging participants to think like an attacker.

6. Consider how and when to use deception to hide the true purpose of your study

   In some cases, informing participants of the true nature of a study could bias them to behave differently than they would in a similar real-life scenario. For example, participants may be unwilling or ashamed to admit they do not consider security. However, the use of deception significantly complicates the study design and in general, the research should strive to be as transparent with participants a possible (as discussed in the next guideline). Thus, researchers should balance these needs by considering when deception is necessary to maintain the study's integrity.

7. Consider ethics in the design of the study

   It is morally imperative that experiments on human subjects be performed ethically. Therefore, it is important that the researcher consider potential ethical dilemmas during the study's design. For example, studies of habituation to compiler security warnings could cause developers to be more likely to ignore these types of problems in their normal work. Researchers should consider these possible side effects and design mitigations against them. The principles of the Belmont Report–*Respect for Persons*, *Beneficence*, and *Justice*–and the Menlo Report, which builds on the Belmont report focusing on information and communications technologies, provide useful guidelines for ethical considerations.

4.2.1.1 Future work

Future work will look at refining this list and providing further specific design examples for secure development studies. We will look at collecting these guidelines both from seminar participants and the broader community into an online living document. This document could provide a useful reference point for researchers new to the field.

Table 1 Open questions for secure software development study methodology.

Question	Amount Currently Known
How much specification or documentation should be provided for each task? Should unit tests be provided?	○
How much prompting for the importance of security should be provided?	◐
How much time should be provided to complete the task? A fixed time budget or unlimited time?	○
Task complexity and fidelity to real-world tasks:	
Reading code vs. writing code	○
Real-world code vs. contrived experimental code	○
From-scratch development vs. editing pre-written code	○
Should participants receive feedback and then get a second chance to improve their code?	◐

4.2.2 Studying Methodology

The second group considered open questions with regard to study design for secure development. The goal was to brainstorm empirical studies that could shed light on how to make appropriate study-design choices in secure-development studies.

Table 1 reviews the open questions the group considered. For each question, the group characterized what is currently known about this question: a significant amount (●), some (◐), or little to none (○).

Further discussion on the general idea of how to think about task instructions and fidelity suggested several potential concrete studies:

- What does real-world tasking in industry look like (interview study)
- Comparing writing secure code to fixing insecure code
- Adding irrelevant details to the task specification to increase cognitive load
- Comparing different amounts of security framing and/or role-playing, to similar tasks without context
- Comparing convincing deception, unconvincing deception, and no deception
- Comparing starting from scratch to adding new features, with different size additions
- Comparing results when task code is written on paper, with a text editor, with an IDE, and with a debugger
- Comparing requiring participants to use a standard development setup to allowing them to use their own preferred setup
- Repeating any of the above studies across a variety of security contexts (e.g., not just cryptography)

4.2.3 Metrics for Studying Secure Development

The final group explored features of developers and the organizations they belong to that might be useful in future work. Such features could be measured as outcome variables, or used as covariates in study design. For each identified feature, the group sought to identify existing metrics used to measure these features. This brainstorming exercise highlighted several metrics that could be adopted by secure development researchers. For example, this included measurements for general security awareness (Human Aspects of Information Security Questionnaire (HAIS-Q) [11]), security behavior (Security Behavior Intention Scale (SeBIS) [7]), and developer personality (Five-Factor Personality Inventory [12], Consideration for Future Consequences (CFC) [10], Need for Cognition (NFC) [9], Domain-Specific Risk-Taking scale (DoSpeRT) [8]). The group also suggested further investigation of psychology, sociology, and anthropology literature to identify metrics for features such as emotional state, frustration towards security, and organizational culture. Finally, the group identified some features that have historically only been measured using simple, Likert-scale self-report questions. This included expertise in development and security. Some work has attempted to use knowledge assessments to gauge these expertise levels, but these tests tend to be cumbersome, and there is no clear evidence of their validity for measuring expertise beyond the specific questions asked. Future work should consider more efficient alternatives.

References

1 Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Proceedings of the 13th Symposium on Usable Privacy and Security (SOUPS)*, pages 81—95, 2017.

2 Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith Deception Task Design in Developer Password Studies: Exploring a Student Sample. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS)*, pages 297—313, 2018.

3 Alena Naiakshina, Anastasia Danilova, Eva, Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. "If You Want, I Can Store the Encrypted Password": A Password-Storage Field Study with Freelance Developers. In *Proceedings of the 37th Conference on Human Factors in Computing Systems (CHI)*, pages 140—152, 2019.

4 Forrest Shull, Janice Singer, and Dag I.K. Sjøberg. Guide to Advanced Empirical Software Engineering. *Springer-Verlag*, 2007.

5 Stuart Schechter. Common Pitfalls in Writing about Security and Privacy Human Subjects Experiments, and How to Avoid Them. *ht-tps://cups.cs.cmu.edu/soups/2010/howtosoups.pdf*, 2010.

6 Helen Nissenbaum. Privacy as Contextual Integrity. *Washington Law Review*, pages 119–157, 2004.

7 Serge Egelman and Eyal Peer. Scaling the Security Wall: Developing a Security Behavior Intentions Scale (SeBIS). In *Proceedings of the 33rd Conference on Human Factors in Computing Systems (CHI)*, pages 2873–2882, 2015.

8 Ann-Renee Blais and Elke U. Weber. A Domain-Specific Risk-Taking (DoSpeRT) Scale for Adult Populations. *Judgement and Decision Making*, vol. 1, no. 1, pages 33–47, 2006.

9 John T. Cacioppo, Richard E. Petty, and Chuan Feng Kao. The Efficient Assessment of Need for Cognition. *Journal of Personality Assessment*, vol. 48, no. 3, pages 306–307, 2010.

10 Jeff Joireman, Monte J. Shaffer, Daniel Ballet, and Alan Strathman. Promotion Orientation Explains Why Future-Oriented People Exercise and Eat Healthy: Evidence From the Two-

Factor Consideration of Future Consequences-14 Scale. *Personality and Social Psychology Bulletin*, vol. 38, no. 10, pages 1272–1287, 2012.

11 Kathryn Parsons, Dragana Calic, Malcolm Pattinson, Marcus Butavicius, Agata McCormac, and Tara Zwaans. The Human Aspects of Information Security Questionnaire (HAIS-Q): Two further validation studies. *Computer & Security*, vol. 66, pages 40–51, 2017.

12 Paul Costa and Robert R. McCrae. Revised NEO Personality Inventory (NEO PI-R) and NEO Five-Factor Inventory (NEO-FFI): Professional Manual. *Psychological Assessment Resources*, 1992.

4.3 Clever Recruitment Techniques: How to Design Studies that get Enough of the Right Kind of Participants

Tobias Fiebig (TU Delft, NL), Michael Coblenz (Carnegie Mellon University – Pittsburgh, US), and Fabio Massacci (University of Trento, IT)

Joint work of Luigi Lo Iacono, Sascha Fahl, Michael Coblenz, Tobias Fiebig, Adam J. Aviv, Florian Alt, Tamara Denning, Daniel Votipka, Michelle Mazurek, Fabio Massacci

4.3.1 Introduction

Recruiting participants for studies among professionals is a serious obstacle for studies on programmers' security behavior and practices. In the past, researchers from the usable security community tried to use mass mailing campaigns to app developers [1], community driven approaches [2], and recruiting among students [3]. In this workshop, we investigated recruitment techniques and incentives for professionals, and found that they are tied to the underlying research questions and design of studies on developers' security behavior.

4.3.2 Key-Insights and Research Objectives

The key-insights of our workshop are:

- The right way to recruit depends on the research question of a study, which determines the population we want to draw from, and thereby the channel and incentives we can use for recruitment.
- Our knowledge of developer types (as well as other involved stakeholders) is limited. Hence, future qualitative study should work towards a taxonomy of stakeholders and programmer types in software development.
- Our knowledge of existing recruitment channels and incentive models is mostly incidental. As such, a structured literature study should enable a more informed view on what worked in the past.
- With the currently unclear situation, extensive pre-testing is essential for finding the right combination of recruitment methods and target population. a future study should aim at validating selected combinations of methods and target groups for common research question types.

4.3.3 Common Research Question Structures

First, we explored common research question setups, to explore the context of studies on secure software development. We identified four distinct research question setups:

1. How does a certain artifact/organizational structure/training X promote or not promote better secure outcomes Y?
2. What are the incentives/barriers X to improving security in a particular operational condition Y?
3. What are the challenges/problems faced by population X, and how can they be identified/mitigated/denied?

We make two observations based on these general research questions: (i) While we selected general descriptions, in practice instanciations of these question may have a substantial conceptual overlap, and, (ii) The setup of the research question may already determine the population we have to select our sample from.

4.3.4 Population Types and Roles

Next, we attempted to categories the types and roles involved in producing software.

- Managers
- Programmers as users
- Testers
- Security teams
- Designers
- Domain experts
- Tool creators
- Users
- Programmers as creators
- Architects/Requirements engineer
- Code reviewers
- Hackers
- Attackers

We note that we do not have enough validated insights into the roles commonly involved in software development. The above list is tentatively based on an in-group brainstorming. While, technically, these types and roles can be derived from descriptive literature of the software development process, this does not necessarily reflect how these are set-up in practice. Hence, we suggest further qualitative studies, combined with literature research, to identify types and roles, and validate them in practice.

4.3.5 Incentives

Next, we discussed incentives for study participation. Even though our research could be better informed by, e.g., Motivational Theory from psychology and social science, we note that the field of software development usually changes frequently. Hence, insights from other fields, especially if they are older, may not transfer directly. Incentive methods we discussed are:

- **Cash:** Either as a flat-payment, a bonus for completion, a raffle, per-hour, as a prize, or as additional staff time (if mandated by leadership). A bonus for good performance might mitigate boredom during the study.
- **Fame**, i.e., by combining the survey with a leader board. Competing with others in general (Vanity/Victory).
- Enabling a **rare experiences** and/or Fun for participants, e.g., driving in a race car, joining a key-note, etc.

- The prospect of **helping others**.
- Helping ones' own organization / Oneself.
- Building better (own) system.
- An opportunity for networking, **social gains**, and competence gains, i.e., by obtaining new technical skills or exposure to potential interesting technology.
- **Distraction** from ones work.
- Obtaining **Credits** (students).
- Learning **best practices** for (own) studies.

We note that some of these incentives may indeed be rather expensive. Others, especially unique experiences, may be more readily available to academic researchers than expected. For example, several universities have student teams operating FormulaE or Forzza race cars.

4.3.6 Recruitment Channels

Finally, we discussed recruitment channels.

- *Community engagement:* For example, in-person participation at hacking events, present at industry conferences, using twitter/slack with hashtag #organization, retweeted by the organization. These are difficult to scale and not necessarily sustainable.
- *Going to professional events focusing on recruitment:* This can be supported by snowballing at conference (with quick ways, recording the conversation), by having a booth, or by giving a talk at conference, asking the audience to participate. Furthermore, it might be possible to integrate the survey in the talk, e.g., by using mentimeter (mentimeter.com) with live display during the presentation.
- *Top-down contact within industry/professional organizations:* Letting industry organizations contact their members might be more incentivizing to professionals. This is especially relevant for questionnaires that should be answered by managers. In general, involving management first seems to provide reasonable results.
- *Class projects with optional participation to the study:* This primarily leverages ongoing classes, but one might also use mailing lists for previous courses and advertising through other classes. A participant notes that lots of people give interest but do not follow up on this.
- *Recruit students locally:* For example with tear-off strips, and strategically placed, e.g., in start-up hubs, or via local mailing list (academic organization) and flyers.
- *Crawling public information:* For example, crawling and contacting Github users and Google play (App authors).
- *Hiring:* Via platforms like Mechanical turk and other freelancer platforms.
- *Building a panel of people interested for future studies:* This can be combined by combining going to conference, and asking participants if they are interested in future studies, and getting a bunch of people to keep up with the panel.
- *Engaging Local companies:* For example, by integrating them in thesis/internship programs, and directly recruiting CTOs/CIOs for interviews.
- *Release an application and wait until people actually use it.*
- *Offering Trainings:* This uses "hands-on" as an exercise, building on a within subject design.
- *Go where people are waiting:* If a significant wait time is involved, people might seek some distraction. The entrance queues at developer conferences might be handy here.
- *Facebook Ads:* These adds usually cost a few hundred dollars, and the standard answer rate is around 3%; Completion of the survey may be optional.

- *Send personal invitations:* Mass mails have a specific "smell" to them, which might disincentivize participation. Making them less standardized might induce a sense of personal engagement, leading to higher conversion.

4.3.7 Summary and Further Work

In summary, we find that pre-test and continuous evaluations are necessary for each experimental design. This should be an iterative process with several pre-tests. For further work, we suggest to test what incentives work with which populations for which recruitment techniques. In general, this will not be possible for an exhaustive list. Hence, this study should be informed by the most common research questions and their relevant populations. The study should build on motivation theory from psychology and social science. Similarly, we have to build a taxonomy of populations, i.e., types of developers and other involved stakeholders. This may be further informed by systematizing knowledge on which combinations of populations, recruitment methods and incentives prove successful in the literature.

References

1 Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 289–305, 2016.

2 Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. Investigating System Operators' Perspective on Security Misconfigurations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1272–1289, 2018.

3 Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar Weippl. "I Have No Idea What I'm Doing"-On the Usability of Deploying HTTPS. In *Proceedings of the USENIX Security Symposium (UseSec)*, pages 1339–1356, 2017.

4.4 API Usability Heuristics

Matthew Smith (Universität Bonn and Fraunhofer FKIE, DE) and Joseph Hallett (University of Bristol, GB)

Security API heuristics can give some metrics about what security problems reside in code. The term has slightly different meanings to different communities however, but generally security *principles* or *heuristics* are smaller, more general rules independent of technology, whereas security *guidelines* are larger and more specific, and security *standards* have been agreed upon by a larger community. It is also not entirely clear what a *security API* is, as the term can include not just cryptographic and security related code but also APIs where the security aspects are less obvious; for instance REST APIs and file access dialogues, and data structures.

There are many different problems we might hope a security API heuristic might be able to detect. Problems include issues with *mental models* or *naming conventions* where a developer's assumptions lead them to believe that APIs disallow (or allow) certain behaviour, such as the ability to access local files from a URL interface; issues with *adversarially controlled input* where a developer mistakenly believes that an input will always be sanitised.

Issues with *documentation* are common where documentation is missing or wrong; as well as issues with canonicalization, redundancy, immutability and call order. Even if an API is correct at one time, updates to library APIs can lead to functions and patterns becoming deprecated and needing updates, and building and linking a library is something many developers struggle with.

Security heuristics, such as Green and Smith's rules for a good crypto API suggest principles to help fix these problems but how might we evaluate their effectiveness? A/B testing is hard to use as the sheer number of parameters can lead to unreasonably large numbers of participants and tests being performed. Instead other approaches such as qualitative evaluation of heuristics against APIs, checking known API problems against proposed heuristics, and analysing change logs to see how APIs were adapted in response to a usability problem may provide better results.

Future work will look at collecting API heuristics and guidelines into an online resource, that would also collect new proposals for heuristics and the results of studies using the heuristics. Other work will look at running experiments to see whether students produce more secure APIs when given heuristics than without them. Finally if a study shows that a significant number of people fail to use an API correctly because of a usability issue, then we ought to be able to report and track this issue–perhaps a new *usability* CVE database might be a sensible way forward.

4.5 Software Security Metrics

Eric Bodden (Universität Paderborn, DE)

Joint work of Eric Bodden, Michael Coblenz, Tamara Denning, Shamal Faily, Joseph Hallett, Mike Lake, Riccardo Scandariato, Reinhard Schwarz, Adam Shostack, Laurens Sion, Charles Weir, Laurie Williams

How do we know that a system is really secure? Currently, a common notion seems to be that a software is secure if it is free of vulnerabilities. But given that definition, it is now clear to practitioners that no software is secure, as *some* vulnerabilities will always be present – be it on the architectural level, in the coding or through the execution environment. In this working group we therefore discussed that metrics for software security should take into account that one should *assume breach*, i.e., a secure system must be able to withstand attacks *despite* known or unknown vulnerabilities. This is somewhat akin to the established notion of resilience, which in the past has been mostly associated with safety, not security properties. Security, if defined that way, is not a binary property but instead is a value on a gradual scale: a system can be more secure than another, but it probably cannot be secure per se. While this might sound disappointing, it very much reflects reality.

There are different stakeholders with respect to which such metrics could be established. Software architects and decision makers in software engineering would probably appreciate structural metrics over software artifacts that correlate well with security. Such metrics could include the fraction of *security code*, i.e., code that implements security features, of the overall application code. Another metric would be how many issues (of a given category) were found in a (potentially tool-assisted) code inspection of some given subsystems. This metric could then be weighted depending on the nature of this subsystem.

Some other–potentially more promising–metrics can be defined from an attacker perspective. With respect to the assume-breach paradigm, an interesting metric is "How many vulnerabilities in combination does an attacker need to exploit to fully own the system (and how hard are these to exploit)?" For many current system, this number will be one! In fact, participants in the breakout group hypothesized that every software system has an "Achilles heel" in the sense that there is a single line of code in the system that if disabled or modified all security is off. If this is true then secure systems must do everything in their power to protect such subsystems. Other attack-centric metrics include measuring an "attack surface" by tracking taint flows from inputs to security-critical components. With respect to long-term security, it is interesting to measure how easily a potentially or likely vulnerable component (say, a complex parser) could be hardened or replaced.

With these metrics as with all others it is currently an interesting and open research questions of how well they correlate with a sensible notion of security. It thus seems that the community would need to establish...

1. an agreed-upon, well-defined and measurable notion of security
2. a set of well-defined metrics with clear descriptions that metrics yield reproducible results
3. a number of empirical studies that seek to correlate these metrics (2) with the notion of security from (1)

Those metrics that correlate best could then yield effective software security metrics.

4.6 Methods For Empirical Studies of SDLs

Sam Weber (Carnegie Mellon University – Pittsburgh, US), Adam J. Aviv (U.S. Naval Academy – Annapolis, US), Michael Coblenz (Carnegie Mellon University – Pittsburgh, US), Tamara Denning (University of Utah – Salt Lake City, US), Shamal Faily (Bournemouth University, UK), Mike Lake (CISCO Systems – Research Triangle Park, US), Steven B. Lipner (SAFECode – Seattle, US), Michelle L. Mazurek (University of Maryland – College Park, US), Xinming (Simon) Ou (University of South Florida – Tampa, US), Olgierd Pieczul (IBM Research – Dublin, IE), Charles Weir (Lancaster University, UK)

In this breakout group we discussed appropriate methodologies for empirically studying secure development processes. It would be ideal if we, as a community, could produce a guide to researchers on what techniques and methodologies to use in order to empirically evaluate competing processes for secure development.

Unfortunately, the current state of the art falls far short of where we need to be in order to produce such a guide. Even such apparently easy process changes, like removing a static analysis from the acceptance tests, turn out in practice to be surprisingly subjective and highly context-dependent. Other changes are harder to evaluate: if we see more defects, are we creating more or are we better at detecting them? It was clear to the group that effective research methodologies was itself a non-trivial research topic.

In order to make a more concrete process, we discussed both hypotheses about secure development that the community maintains, but would like more concrete evidence for, and what the important outstanding research questions are. We then did a deep-dive on one of them (threat-modeling).

Things that the community believes are true, but which could use empirical evidence, include:

- threat-modeling is very important,
- vulnerabilities and vulnerability mitigation tasks are distinct from each other and need to be tracked as such in the bug tracking system, and
- securing third-party code/libraries is critically important.

Outstanding research questions (many of which are related to the above) include:

- the costs and benefits of security-related choices like the use of type-safe languages,
- the impact of various software architectures on security,
- what are the key advantages of threat modeling? Finding real threats, supporting communication, focusing thinking, allowing non-security-experts to have input into the process?
- When deciding whether or not to incorporate third-party code, what due-diligence processes are effective?
- how do processes like penetration testing affect company security culture, and how to distinguish the effects of community culture from specific processes?

In our threat-modeling deep-dive, we realized that there were a myriad of desirable properties of a threat modeling methodology, such as being low-cost, or easy to use by non-security experts. There are also many possible desirable properties of a threat model, such as identifying only feasible attacks, or enabling estimating of the cost of exploitation. A number of experimental methods for evaluating different threat modeling methodologies were brought up, including

- Obtaining a "gold standard" for a given scenario and comparing the results produced by each method,
- Ethnographic methods, where real-world teams are observed,
- Leverage the training process of companies that are bringing in external threat-modeling trainers and
- Conducting internal analyses, where an organization collects threat-modeling artifacts and then later reviews them versus vulnerabilities that were found or missed. As the actual threat model is likely to be sensitive, it probably cannot be released, but the comparison should be able to be disclosed.

The group agreed that none of these methods were "better" than the others, but rather that all of them obtain different views of threat modeling. By using disparate techniques a more complete picture can be formed. Although time prevented us from doing similarly deep dives on other problems, the group felt that this was probably a general observation: given the nature of cybersecurity, using a variety of techniques will be almost certainly required to give a complete picture of the benefits and drawbacks of any particular development methodology.

4.7 Publication or reviewing guidelines; Establishing a baseline for evidence of science in security

Laurie Williams (North Carolina State University – Raleigh, US)

To build a science, researchers need to document scientific evidence of their work so that future researchers can support or evolve the hypotheses and theories documented in published reports and papers. Carver et al. analyzed papers that appeared in two top security conferences, ACM Computer and Communications Security (CCS) and IEEE Security and Privacy (S&P) [1, 2]. Their main motivation was to assess whether the papers reported information necessary for three key pillars of scientific research: replication, meta-analysis, and theory building. To perform this work, Carver's team utilized a rubric with which they analyzed the papers: http://carver.cs.ua.edu/Studies/SecurityReview/Rubric.html. The group discussion revolved around looking at the components of this rubric and discussed the benefits of providing rubrics to enable researchers to publish papers that contain the scientific evidence others can build upon.

References

1 Morgan Burcham, Mahran Al-Zyoud, Jeffrey C. Carver, Mohammed Alsaleh, Hongying Du, Fida Gi- lani, Jun Jiang, Akond Rahman, özgür Kafalı, Ehab Al-Shaer, and Laurie Williams. *Characterizing scientific reporting in security literature: An analysis of ACM CCS and IEEE S&P papers* In Proceedings of the Hot Topics in Science of Security: Symposium and Bootcamp, HoTSoS, pages 13–23, New York, NY, USA, 2017. ACM.

2 Jeffrey C. Carver, Morgan Burcham, Sedef Akinli Kocak, Ayse Bener, Michael Felderer, Matthias Gander, Jason King, Jouni Markkula, Markku Oivo, Clemens Sauerwein, and Laurie Williams. *Establishing a baseline for measuring advancement in the science of security: An analysis of the 2015 IEEE Security & Privacy proceedings* In Proceedings of the Symposium and Bootcamp on the Science of Security, HotSos '16, pages 38–51, New York, NY, USA, 2016. ACM.

Participants

- Florian Alt
Universität der Bundeswehr – München, DE
- Adam J. Aviv
U.S. Naval Academy – Annapolis, US
- Eric Bodden
Universität Paderborn, DE
- Michael Coblenz
Carnegie Mellon University – Pittsburgh, US
- Tamara Denning
University of Utah – Salt Lake City, US
- Serge Egelman
ICSI – Berkeley, US
- Sascha Fahl
Leibniz Universität Hannover, DE
- Shamal Faily
Bournemouth University, GB
- Tobias Fiebig
TU Delft, NL
- Joseph Hallett
University of Bristol, GB
- Trent Jaeger
Pennsylvania State University – University Park, US
- Mike Lake
CISCO Systems – Research Triangle Park, US
- Carl E. Landwehr
George Washington University – Washington, US
- Steven B. Lipner
SAFECode – Seattle, US
- Luigi Lo Iacono
FH Köln, DE
- Fabio Massacci
University of Trento, IT
- Michelle Mazurek
University of Maryland – College Park, US
- Brendan Murphy
Microsoft Research – Cambridge, GB
- Brad A. Myers
Carnegie Mellon University – Pittsburgh, US
- Xinming (Simon) Ou
University of South Florida – Tampa, US
- Olgierd Pieczul
IBM Research – Dublin, IE
- Heather Richter Lipford
University of North Carolina – Charlotte, US
- Riccardo Scandariato
Chalmers and University of Gothenburg, SE
- Reinhard Schwarz
Fraunhofer IESE – Kaiserslautern, DE
- Adam Shostack
Seattle, US
- Laurens Sion
KU Leuven, BE
- Matthew Smith
Universität Bonn and Fraunhofer FKIE, DE
- Walter F. Tichy
KIT – Karlsruher Institut für Technologie, DE
- Daniel Votipka
University of Maryland – College Park, US
- Sam Weber
Carnegie Mellon University – Pittsburgh, US
- Charles Weir
Lancaster University, GB
- Laurie Williams
North Carolina State University – Raleigh, US
- Mary Ellen Zurko
MIT Lincoln Laboratory – Lexington, US