Panel: Empirically-based Secure OS Design

Sam Weber
samweber@acm.org

Adam Shostack
adam@shostack.org

Jon A. Solworth
solworth@rites.uic.edu

Mary Ellen Zurko
mez@alum.mit.edu

ABSTRACT

This NSPW panel discussed how we, as a community, should pursue evidence-based research on designs for commoditizable mass-market secure operating systems. This panel did not discuss *what* features or architectures should be adopted, but instead focused on *how* we evaluate competing features and designs. How can, or should, we gather data about the usability and resilience of secure OS designs without requiring massive implementations and deployments?

CCS CONCEPTS

• **Security and privacy → Operating systems security**;

KEYWORDS

OS Security, science

ACM Reference format:
Sam Weber, Adam Shostack, Jon A. Solworth, and Mary Ellen Zurko. 2017. Panel: Empirically-based Secure OS Design. In *Proceedings of New Security Paradigms Workshop 2017, Santa Cruz, California, USA, October 1-4, 2017 (NSPW'17),* 4 pages.
DOI: 10.1145/3171533.3171543

1 INTRODUCTION

The design of secure operating systems is one of the oldest topics in cybersecurity. Unfortunately, since Multics, there has been an enormous divide between the research community and the commodity OS vendors, and we all pay the cost of the resulting attacks. This should not be interpreted as criticism of OS venders — Microsoft in particular has made great strides in incorporating security into Windows — but rather a statement that our community has difficulties in transitioning its research to practice.

Our community has a plethora of ideas about features and architectures that would contribute towards a more secure operating system, but there is little empirical evidence to which combination of features and architectures would lead to a *usable* OS which has enough functionality to compete with existing popular operating systems. It is our belief that this is a major cause of our poor history of transition to practice. It is one thing to show that a small research OS is able to securely execute a small command-line program, but another to demonstrate that it would be able to securely and usably host a modern spreadsheet application which includes a powerful macro language able to execute queries on remote databases and support third-party plugins. It is only to be expected that a commercial OS vendor will be reluctant to adopt a major departure from established practice without considerable evidence.

Alas, this isn't just an academic exercise. Every once in a while a disruptive event happens in which an opportunity for a new OS arises. The introduction of smartphones was one such recent opportunity. When such an event arises, practitioners do not have time to undertake lengthy research projects to come up with an ideal design, but have to quickly adopt the best-argued ideas that come to hand. The multitude of sessions on mobile security in the major cybersecurity conferences demonstrate that our community failed the smartphone OS designers: could we not have antipated and prevented many of the security flaws of, for instance, Android, *before* they were inflicted on millions of users? Can we be prepared the next time an opportunity takes place?

Another avenue which increases the likelihood of adoption is Virtual Machines (VMs) which enable more specialized OSs to coexist with more general OSs on the same hardware. Operating Systems such as Qubes [3] and before that VMM [1] use VMs to provide security properties beyond what their supported operating systems do.

To be clear, this panel was not focused *what* features, designs or architectures should be part of a secure OS: we do not want to take a stance on capabilities vs access-control lists, various information flow mechanisms, or any other such technical feature. Instead, we discussed *how* we should conduct empirically-based research in this field. It is easier to do proofs that a particular mechanism or feature implements some formal policy than to determine whether regular users can use it, or whether developers can effectively build secure enterprise-scale applications on top of it. How can, or should, we gather data about secure OS designs without requiring massive implementations and deployments?

2 SCOPE OF PANEL

In order to be productive and to avoid endless side discussions we scoped the topic of this discussion. Firstly, we wished to only discuss research methods, not the benefits or lack thereof of various security features or architectures. For example, we didn't want to discuss the relative merits of capability systems versus ACLs. However, metrics by which one could compare capabilities and ACLs, and experimental designs to evaluate them, were a valid discussion topic.

Secondly, we defined what we mean by a reasonable commoditizable operating system. There is a tendency to consider highly constrained systems, making security easier. Although there are many real-world instances of such systems, considering them would make the discussions less interesting.

An operating system is supposed to provide services to users and applications. We want to design systems which are supposed to support the general public, running applications such as internet browsers and office applications (like word processors and spreadsheets). We also want to consider web application servers and databases.

Perhaps more controversial is what we meant by "supporting applications". Many OSes define security so that the OS protects only its own resources, and provides minimal support for application developers to secure their applications. To an average user, their applications and associated data is all they care about. The fact that in many systems a compromised application is allowed to do anything that the user can do makes the situation even worse. If a user's tax and health information is stolen because of single application's flaw, it is of little consolation that the OS's security policy was not violated. Even though it is surely impossible for an OS to absolutely prevent application security vulnerabilities (at least for any useful definition), the OS should assist developers to create secure applications. To put it another way, a secure OS is one which provides security services to its users, and application developers are users too.

3 DISCUSSION TOPICS

3.1 Usable and Effective Access Control

Most widely-used commercial operating systems use variations of the classic access control model in which "subjects" are given rights over "objects" in the system, and where programs are implicitly given the rights of the application's user. We know that this model is quite inadequate for the modern world: a spreadsheet is hardly a passive entity when it contains an executable script, and it is not clear if the spreadsheet application is continuing to operate on behalf of the user when it is executing that script. Alternative models exist. Capability systems, in particular, are quite good at capturing information flow. However, whether these systems are mathematically better is only part of the story. It is also necessary to judge whether users and programmers are able to effectively understand and use these models. What evidence can we gather to evaluate the usability of competing access-control models?

In particular, capability systems have had a long history of been supported by various academic arguments: they correspond better to information flow than ACLs do, they allow one to implement confinement, and so on. However, generally they've not met with commercial success, except in less global ways (ie, file handles in many filesystem APIs are essentially capabilities). Why is this so? Is it because programmers and system architects are simply unfamiliar with capabilities, and therefore stick to what they know? Is it because ACLs better correspond to their mental models? Or, in practice, do programmers using capability system end up having to manage a confusing pile of capabilities, making the problem no easier?

There are difficulties in trying to gather empirical evidence to answer these questions. Most lab studies end up being quite time-limited, due to budget concerns. As a result, there is the concern that lab studies would essentially just measure learnability. Would case studies of small or medium-scale systems be persuasive? And the effect of designs on security is usually not immediately apparent, so how long would a study have to be conducted before reliable conclusions could be drawn?

During the discussion it was noted that this was a topic that frequently is brought up: one of the participants, for example, has been in many meetings where capabilities were proposed to handle fine-grain access control, but others argued that it was not possible to manage least privilege and access control at scale in a capability-based system. Microsoft has reportedly implemented a capability-based system called Midori, but it hasn't been released. Unfortunately, it is not known whether its non-release is due to technical issues or merely political or commercial judgements.

3.2 Composition and Decomposition

An operating system, like a programming language, is a base for creating and running applications. For applications there are many external requirements, which restrict the form of applications. But for an operating system there is only one question: What is the best form to create applications with the appropriate properties? These properties include Security, Privacy, and Usability (which themselves are composed of many sub properties).

To address this rather unconstrained problem, it is necessary to decompose the problem into components, hone the design of these components, test them against the various desired properties, and study how they compose back to provide solutions. It is in this composition that programs often fail, so a careful design is necessary but not sufficient.

Ultimately, the proof is in its use: How difficult is it to build applications with the desired properties? What are the limitations of the system? Do the components work well together in the different ways they must compose? These are not questions which can be answered with proofs. Instead, empirical evaluation is essential.

3.3 OS Support for Applications

Operating systems are supposed to support applications. One way that they do so is via the APIs and programming model that they implement. We have many examples of APIs that lead to insecure code: a file system API that provably cannot be free of TOCTOU vulnerabilities is hardly ideal. Can such properties be provably provided, or are we limited to features which if properly used are safer, but not guaranteed? We are woefully lacking empirical evidence of how to design better APIs and models. This is especially true when it comes to complex applications such as web application servers and internet browsers which have to implement complex security policies of their own—what support can an OS provide to applications such as these?

This is not to say that there has not been work on how to design OSes that help developers write more secure code. As one example, the Ethos [7] project has this as its major goal. Rather, the question is how to evaluate the effectiveness of any particular design or approach. Evaluating the usability of an API is hard enough, but evaluating whether it leads programmers to write more secure code is harder, especially as there is no established metric for security. Perhaps a decomposition of properties will help with this evaluation?

3.4 Learning from Practice

SELinux [4] is a widely-available security extension to Linux. Internet browsers are now often designed as if they were operating systems and implement their own security mechanisms. Java itself also might be considered to provide OS support to its applications. What lessons can be learned about secure OS design from these systems?

For example, SELinux is often derided as being hard to use. Does this represent an intrinsic problem with the protection model it implements? Or is it merely, as Chris Siebenmann suggests [6], the fact that it is not the primary security model of the system but instead a secondary mechanism? As for Java, Java's classloading mechanism ended up being used entirely differently than anticipated by its designers (as a modularity construct, rather than a security mechanism). What does this imply about the software ecosystem that a security system should support?

Our community seems to have difficulty in learning from real-life examples such as these. Can we do better?

3.5 Evaluating Security of Applications On an OS

If we want to assess if we have a secure OS design that supports secure applications, an initial consideration is whether secure applications be created for this OS? The OS may or may not have a model of what a secure application is (e.g. isolation). The application itself may bring in security features which may have full or partial OS support (e.g. authentication, encrypted communications such as HTTPS.). It may bring in features traditionally associated with security bugs or vulnerabilities (e.g. an interpreter). It may call OS features with the same history (e.g. command line). The availability of the application itself may be considered a security feature.

So, to (empirically) create a secure application, we need to start with some choices; development language and tools, (security) features, calling APIs. The argument was made that empirical evidence that the application is secure must include testing, even for provably secure code, since the gap between what can be proven and what is considered secure in use remains. There is a raft of security testing tools and procedures that are considered standard best practice:

- Static analysis if the language warrants it and if the tools are fairly practical (e.g. low false positives, actionable suggestions for fixing issues),
- Fuzz testing of any inputs. Web vulnerabilities (e.g. XSS, XSRF, SQL injection),
- Various kinds of scans, such as Nessus, which can determine if versions of code with known vulnerabilities needs patching or updating,
- the patches/update model that is now commonly used by operating systems, and
- manual pen testing, which is the first form of security testing for any new and innovative technologies or approaches that are not suitably exercised by existing tools and techniques.

If it is posited and shown that it is possible to create secure applications, a next question is, can developers create secure applications on this OS? Empirical research of developers and secure coding is a rapidly emerging area in the usable security community. The community is still sorting through a large number of ecological validity questions, which only loom larger for new and innovative approaches to secure applications and OSes. What level of experience do the developers have with the tools and languages, with the APIs, with the OS? What tasks/applications are you evaluating their ability to code securely? What about developers who work in teams, or in organizations, with the collaboration and processes inherent in multi-developer work? Early research in this area has shown that examples and documentation have the most impact. This is great news for initial testing in this area. Create the documentation and (secure) examples first, and start testing with those.

3.6 'Security' isn't precise enough

Security is not a single goal, but a set of goals, because we lack useful ways to conceptualize what a secure OS does. Let's assume that Apple prioritizes anti-malware in iOS. But one of their features is to restrict what code runs as root; perhaps there is a lot of malware on the platform, but our inability to run AV software on it inhibits our ability to test that hypothesis? The OS is so secure it even resists science!

Similarly, we might claim that the way to empirically evaluate a how well a phone succeeds at anti-censorship might be to test its capabilities in avoiding censorship. That depends in part on the software loaded on it, but it also depends on the censorship, and that censorship might change in response to features in the phone. If we wish to be empirical, which facts should we use?

To follow from Petroski's core argument in *The Evolution of Useful Things*, [2], engineers work to address flaws they observe in real systems. It may be hard or impossible to understand the use of a thing until it's built and deployed, and it may be harder or more impossible to understand the security of a thing in the same fashion. A longer discussion of this is available at [5].

3.7 Miscellaneous Observations

A number of observations were made by NSPW participants that don't fit neatly into the above topics.

It was observed that very few of the problems discussed in NSPW papers would be solved by a secure operating system, which raises the question of whether we need a different abstraction instead. It was argued that whatever different mechanism was desired, it would probably look like an OS. In many cases currently, what runs at the bottom of the software stack is really a browser, not a traditional OS, but modern browsers are, in reality, operating systems.

It was then argued that what an OS provides is harm reduction: after a developer does something bad, an OS can limit the blast radius of the failure. The use of secure development practices, such as the use of Go instead of C, was described by another participant as also harm reduction.

The panelists were asked what the biggest question or lowest-hanging fruit for empirical measurement is. One panelist argued that we need a model better than subject-object-action. Another panelist wanted to know how computers get compromised: what is the proximate event that gives an attacker control of a computer? There are no statistics on this, but the argument is that with this knowledge we could have a more informed idea of what the most impactful design choices were.

4 PANEL ORGANIZATION AND PARTICIPANTS

The panel members were:

Adam Shostack Adam is a consultant, entrepreneur, technologist, author and game designer. He's a member of the BlackHat Review Board, and helped found the CVE and many other things. He's currently helping a variety of organizations improve their security, and advising and mentoring startups as a Mach37 Star Mentor. While at Microsoft, he drove the Autorun fix into Windows Update, was the lead designer of the SDL Threat Modeling Tool v3 and created the "Elevation of Privilege" game. Adam is the author of "Threat Modeling: Designing for Security," and the co-author of "The New School of Information Security."

Jon Solworth Jon Solworth is interested in the design and engineering of robust systems which—unlike today's systems—can withstand attack. His projects in this area include authorization, authentication, denial of service, and what is becoming my capstone project, a new operating system called Ethos.

Notable achievements within computer security include:

1. Designing and leading the development of the Ethos Operating System, with MinimaLT network protocol, whose semantics make it far easier to develop secure applications.

2. Showing that an authorization model (aka) could be both sufficiently expressive to provide desirable protections while also be analyzable, so that the protections could be understood. This solved, in part, a 30 year old problem posed by Harrison-Ruzzo-Ullman.

3. Showing how to do Public Key Infrastructure Revocations 1000s of times more efficiently than the standard technique (OCSP). The Revocation Problem has been called a Grand Challenge Problem in PKI.

Sam Weber Sam Weber's primary research interests lie in the empirical evaluation of secure development methodologies. He obtained his PhD from Cornell University on specification and verification, and he served as a Program Director for the National Science Foundation's Secure and Trustworthy Cyberspace program. He has been a researcher at IBM's T.J. Watson Research Center and a faculty member at Cornell University, the University of Pennsylvania, and New York University.

Mary Ellen Zurko Mary Ellen Zurko has recently joined MIT Lincoln Laboratory. Mez has worked extensively in security; in product development, early product prototyping, and in research, and has over 20 patents. She was security architect of one of IBM's earliest clouds; SaaS for business collaboration. She defined the field of User-Centered Security in 1996. As a senior research fellow at the Open Group Research Institute, she led several innovative security initiatives in authorization policies, languages, and mechanisms that incorporate user-centered design elements. She started her security career at DEC working on an A1 VMM, on which she coauthored a retrospective with a fellow member of the Forum on Cyber Resilience. She has written on active content security, public key infrastructures, distributed authorization, user-centered security, and security and the web. She is a contributor to the O'Reilly book "Security and Usability: Designing Secure Systems that People Can Use." She is on the steering committees of New Security Paradigms Workshop and General Chair of the Symposium on Usable Privacy and Security. Mez received S.B and S.M. degrees in computer science from MIT.

5 CONCLUSIONS

In summary, during this panel we discussed how to effectively apply empirical methods to the problem of designing secure operating systems. Given the long history of cybersecurity in general, and secure operating system research specifically, it is long-past time that we establish an evidence-based body of knowledge on how to create operating systems that are usable, useful, and which assist developers to create secure applications. Furthermore, we do not wish to miss opportunities to introduce secure OS designs into commercial practice.

Acknowledgements. This material is partially based upon work supported by the National Science Foundation under Grant Numbers 1409868 and 1640732.

REFERENCES

[1] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. 1991. A Retrospective on the VAX VMM Security Kernel. *IEEE Transactions on Software Engineering* 17, 11 (1991), 1147–1165. Special Section on Security and Privacy.
[2] Henry Petroski. 1992. *The evolution of useful things.* Vintage.
[3] Qubes OS Project. 2017. Qubes OS: A Reasonably Secure Operating System. http://www.qubes-os.org. (2017). Accessed: 2017-12-15.
[4] SELinux. 2017. SELinux Project Wiki. (2017). https://selinuxproject.org/page/Main_Page
[5] Adam Shostack. 2015. The Evolution of Secure Things. (2015). https://adam.shostack.org/blog/2015/11/the-evolution-of-secure-things/
[6] Chris Siebenmann. 2011. One of SELinux's problems is that it's a backup mechanism. (2011). https://utcc.utoronto.ca/~cks/space/blog/linux/SELinuxIsABackup
[7] Jon Solworth. 2017. The Ethos Operating System. (2017). https://www.ethos-os.org/index.html