AFRL-OSR-VA-TR-2015-0149

Information Flow Integrity for Systems of Independently-Developed Components

Trent Jaeger
PENNSYLVANIA STATE UNIVERSITY

06/22/2015
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTC
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-06-2015	Final Report	April 1, 2012 - March 31, 2015

4. TITLE AND SUBTITLE

Information Flow Integrity for Systems of Independently-Developed Components

5a. CONTRACT NUMBER
FA9550-12-1-0166

5b. GRANT NUMBER
FA9550-12-1-0166

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Trent Jaeger, Penn State University
Vinod Ganapathy, Rutgers University
Somesh Jha, University of Wisconsin-Madison

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Penn State University, University Park, State College, PA 16801
Rutgers University, 190 University Ave, Newark, NJ 07102
University of Wisconsin-Madison, Madison, WI 53706

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research
Wright-Patterson Air Force Base

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution is Unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The aim of this project is to enable enforcement of integrity safe in systems of independently-developed components. In this project, we explore this problem from three perspectives. First, we developed integrity safety properties and mechanisms to enforce them. In particular, we developed resource retrieval (access) integrity, which protects programs when retrieving system resources, and implemented the process firewall mechanism to enforce this property. Second, we developed integrity safety mechanisms for a variety of software, including web browsers (to protect them from browser extensions), kernel software (to enforce resource retrieval integrity and fine-grained control-flow integrity of approved code), and user-space programs (to enforce access control policies). Third, we developed methods to retrofit software to enforce integrity safety properties mostly-automatically through safety games and authorization constraints. Both of these methods enable an efficient deployment of code to enforce expected integrity requirements. This work has been published several top conferences in computer security and programming languages and some

15. SUBJECT TERMS

Computer security, integrity, confused deputy attacks, code reuse attacks, retrofitting legacy code, web browser, operating system kernels, attack surface, safety games, access control, privilege separation, information flow, control-flow integrity

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Trent Jaeger
U	U	U	SAR	8	19b. TELEPHONE NUMBER *(Include area code)* 814-865-1042

Reset

INSTRUCTIONS FOR COMPLETING SF 298

1. REPORT DATE. Full publication date, including day, month, if available. Must cite at least the year and be Year 2000 compliant, e.g. 30-06-1998; xx-06-1998; xx-xx-1998.

2. REPORT TYPE. State the type of report, such as final, technical, interim, memorandum, master's thesis, progress, quarterly, research, special, group study, etc.

3. DATES COVERED. Indicate the time during which the work was performed and the report was written, e.g., Jun 1997 - Jun 1998; 1-10 Jun 1996; May - Nov 1998; Nov 1998.

4. TITLE. Enter title and subtitle with volume number and part number, if applicable. On classified documents, enter the title classification in parentheses.

5a. CONTRACT NUMBER. Enter all contract numbers as they appear in the report, e.g. F33615-86-C-5169.

5b. GRANT NUMBER. Enter all grant numbers as they appear in the report, e.g. AFOSR-82-1234.

5c. PROGRAM ELEMENT NUMBER. Enter all program element numbers as they appear in the report, e.g. 61101A.

5d. PROJECT NUMBER. Enter all project numbers as they appear in the report, e.g. 1F665702D1257; ILIR.

5e. TASK NUMBER. Enter all task numbers as they appear in the report, e.g. 05; RF0330201; T4112.

5f. WORK UNIT NUMBER. Enter all work unit numbers as they appear in the report, e.g. 001; AFAPL30480105.

6. AUTHOR(S). Enter name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. The form of entry is the last name, first name, middle initial, and additional qualifiers separated by commas, e.g. Smith, Richard, J, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES). Self-explanatory.

8. PERFORMING ORGANIZATION REPORT NUMBER. Enter all unique alphanumeric report numbers assigned by the performing organization, e.g. BRL-1234; AFWL-TR-85-4017-Vol-21-PT-2.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES). Enter the name and address of the organization(s) financially responsible for and monitoring the work.

10. SPONSOR/MONITOR'S ACRONYM(S). Enter, if available, e.g. BRL, ARDEC, NADC.

11. SPONSOR/MONITOR'S REPORT NUMBER(S). Enter report number as assigned by the sponsoring/ monitoring agency, if available, e.g. BRL-TR-829; -215.

12. DISTRIBUTION/AVAILABILITY STATEMENT. Use agency-mandated availability statements to indicate the public availability or distribution limitations of the report. If additional limitations/ restrictions or special markings are indicated, follow agency authorization procedures, e.g. RD/FRD, PROPIN, ITAR, etc. Include copyright information.

13. SUPPLEMENTARY NOTES. Enter information not included elsewhere such as: prepared in cooperation with; translation of; report supersedes; old edition number, etc.

14. ABSTRACT. A brief (approximately 200 words) factual summary of the most significant information.

15. SUBJECT TERMS. Key words or phrases identifying major concepts in the report.

16. SECURITY CLASSIFICATION. Enter security classification in accordance with security classification regulations, e.g. U, C, S, etc. If this form contains classified information, stamp classification level on the top and bottom of this page.

17. LIMITATION OF ABSTRACT. This block must be completed to assign a distribution limitation to the abstract. Enter UU (Unclassified Unlimited) or SAR (Same as Report). An entry in this block is necessary if the abstract is to be limited.

Grant/Contract Title: Information Flow Integrity for Systems of Independently-Developed Components

Grant/Contract Number: FA9550-12-1-0166
PI: Trent Jaeger, Penn State University
Date: June 1, 2015

Abstract

The aim of this project is to enable enforcement of integrity safe in systems of independently-developed components. In this project, we explore this problem from three perspectives. First, we developed integrity safety properties and mechanisms to enforce them. In particular, we developed resource retrieval (access) integrity, which protects programs when retrieving system resources, and implemented the process firewall mechanism to enforce this property. Second, we developed integrity safety mechanisms for a variety of software, including web browsers (to protect them from browser extensions), kernel software (to enforce resource retrieval integrity and fine-grained control-flow integrity of approved code), and user-space programs (to enforce access control policies). Third, we developed methods to retrofit software to enforce integrity safety properties mostly-automatically through safety games and authorization constraints. Both of these methods enable an efficient deployment of code to enforce expected integrity requirements. This work has been published several top conferences in computer security and programming languages and some of the projects have been packaged for open-source distribution.

1 Overview

The aim of this project was to develop methods to improve the integrity safety of programs. That is, when a program performs a security-sensitive operation, such as a system call, an indirect control transfer, or an instruction that modifies critical program data, can we define and enforce properties that protect the integrity of those operations.

The team of Dr. Trent Jaeger (Penn State, PI), Dr. Vinod Ganapathy (Rutgers), and Somesh Jha (Wisconsin, Madison) developed several methods to enhance the integrity safety of programs. First, we developed new principles for enhancing integrity safety. We defined principles to ensure that: (1) only approved kernel code is executed, even when the kernel is compromised; (2) each system resource access to either protected from adversary tampering or is restricted to adversary-accessible resources; (3) the retrofitting of programs cannot enable an adversary to control the execution of a program in a manner that would violate a safety game or a set of constraints.

Second, we improved the effectiveness on existing system and system development mechanisms by applying the principles above or improving the application of known integrity safety principles. For example, we extend the traditional OS reference monitoring with a *process firewall* mechanism, which is capable of distinct security policies per each system call to protect program integrity. This mechanism has been optimized to provide system-wide enforcement for low overhead (¡4% over several macrobenchmarks). In addition, we have shown that known integrity defenses, such as control-flow integrity and browser extension confinement, can be leveraged more effectively (in a more automated way) to provide tighter security (finer-grained enforcement).

Third, we have applied integrity safety to several types of software, including kernel software, middleware, and server programs. Kernel software provides a foundation for integrity, and we show that we can both restrict kernel execution to approved code and greatly restrict code reuse attacks on that code (using fine-grained, control-flow integrity). Given integrity protection of the kernel, the process firewall kernel module can enforce integrity safety over the retrieval of individual resources by each program in the system. In addition, protections against browser extensions can protect complex middleware programs. Finally, information flow integrity protections in general can be enforced by retrofitting the program to satisfy integrity constraints, as we have explored using safety games and constraint systems.

Work on kernel integrity, the process firewall, and automated techniques for retrofitting legacy code with security are of broad interest in the research community, so we exploring how to make such methods viable for commercial purposes. As an initial step, we exploring the integration of such mechanisms in conventional kernels, such as Linux, FreeBSD, and Windows, and ecosystems, such as LLVM to enable open-sourcing of such mechanisms. For example, we have released the process firewall for Linux and are planning to release our control-flow integrity enforcing FreeBSD kernel. In addition, Samsung has deployed a version of a mechanism that restricts kernels to approved code that obeys our requirements, as part of their Knox project, and we will likely collaborate with them in the near future. This work has also influenced a new NSF-funded project on retrofitting legacy code for multiple defenses, on which PIs Jaeger and Ganapathy participate.

In the remainder of this report, we review our research results describing the addition of integrity safety defenses in kernel software, middleware, and user-space programs, in one section each.

2 Integrity Safety in Kernel Software

We explored methods to improve integrity safety in kernel software from two perspectives. First, we investigated the problem of protecting the integrity of system resource retrieval. A variety of vulnerabilities, such as link traversal, directory traversal, file squatting, and file system time-of-check-to-time-of-use (TOCTTOU) attacks result from programs being tricked by adversaries to retrieval system resources (e.g., files, IPCs, etc.) chosen by the adversary enable *confused deputy attacks*. Second, we investigated mechanisms to enforce integrity protections on kernel software execution particular to restrict *code reuse attacks*, such as return-oriented programming. We currently trust kernel software to protect its integrity, but a variety of kernel vulnerabilities are motivating the need for integrity protection mechanisms. We developed methods to restrict kernel execution to approved code and restrict that code to execute under the restrictions of fine-grained, control-flow integrity.

2.1 System Resource Retrieval Integrity

In the first work, we investigated the hypothesis that system programs should only interact with adversaries through a small number of program entry points, program instructions that invoke system libraries to retrieve resources (e.g., unique invocations of the `open` library call in each program). The set of entry points of a program that are accessible to an adversary are called its *attack surface*. Experience has shown that developers often fail to defend these entry points because they do not locate all the code locations where programs access system resources controlled by attackers. We developed a runtime analysis method to compute program attack surfaces in system deployments, which uses a novel approach to computing program adversaries to determine which program entry points access adversary-controlled resources. We implemented our approach as a Linux kernel mechanism capable of identifying entry points for both binary and interpreted programs. Using this mechanism, we computed the attack surfaces for all the programs in the Ubuntu Linux 10.04 Desktop distribution automatically. On examining located attack surfaces, we discovered previously unknown vulnerabilities in an X Windows startup script available since 2006 and the GNU Icecat web browser. Our tools enable developers to find attack surfaces for their programs quickly and to produce defenses prior to the emergence of attacks, potentially moving us away from the penetrate-and-patch rut.

"Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies," Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, Trent Jaeger. In Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS), May 2012.

We subsequently found that our mechanism for computing attack surfaces could be used to test whether programs were vulnerable to *resource retrieval attacks*. A resource retrieval attack can occur when a program resolves a system resource name (e.g., file path) into a resource reference. The process of name resolution is fundamental to computer science, but its use has resulted in several classes of vulnerabilities. These vulnerabilities are difficult for programmers to eliminate because their cause is external to the program: the adversary changes namespace bindings in the system to redirect victim programs to a resource of the adversarys choosing. Researchers have also found that these attacks are very difficult to prevent systematically. Any successful defense must have both knowledge about

the system namespace and the program intent to eradicate such attacks. As a result, finding and fixing program vulnerabilities to such as attacks is our best defense. We developed the STING test engine, which finds name resolution vulnerabilities in programs by performing a dynamic analysis of name resolution processing to produce directed test cases whenever an attack may be possible. The key insight is that such name resolution attacks are possible whenever an adversary has write access to a directory shared with the victim, so STING automatically identifies when such directories will be accessed in name resolution to produce test cases that are likely to indicate a true vulnerability if undefended. Using STING, we found 21 previously-unknown vulnerabilities in a variety of Linux programs on Ubuntu and Fedora systems, demonstrating that comprehensive testing for name resolution vulnerabilities is practical.

"STING: Finding Name Resolution Vulnerabilities in Programs," Hayawardh Vijayakumar, Joshua Schiffman, Trent Jaeger. In Proceedings of the 21st USENIX Security Symposium, August 2012.

We then began to focus on methods to protect programs from resource retrieval attacks. We first explored whether we could infer security policies that would block such attacks using properties of the resources themselves. We developed a dynamic analysis that collected whether adversaries of the program (defined using the ASIACCS 2012 adversary model) could modified files retrieved at each program entrypoint. We found that, by using this approach for Ubuntu 12.04, 98.5% of accesses can be restricted to prevent typical name resolution attacks and more than 65% of accesses can be restricted to a single file without creating false positives. We also examined three programs (Apache, MySQL, and PHP) in detail to evaluate the efficacy of using the provided package test suites to generate policies, finding that administrators can produce effective policies automatically.

"The right files and the right time," Hayawardh Vijayakumar and Trent Jaeger. In Proceedings of the 5th Symposium on Configuration Analytics and Automation (SafeConfig), October 2012.

We next developed a defensive mechanism to prevent adversaries from exploiting program vulnerabilities during resource retrieval, which we call a *process firewall*. The process firewall is a kernel mechanism that protects each system call of a process by introspecting into the current process state (e.g., call stack) and the OS's file system state to prevent processes from retrieving resources that violate integrity rules for the process and OS states. The key insight is that the process firewall only protects processes rather than confining them so it can examine their internal state to identify the protection rules necessary to block many of these attacks without the need for program modification or user configuration. We built a prototype process firewall for Linux, demonstrating: (1) the prevention of several vulnerabilities, including two that were previously-unknown; (2) that this defense can be provided system-wide for less than 4% overhead in a variety of macrobenchmarks; and (3) that it can also improve program performance, shown by Apache handling 3-8% more requests when program resource access checks are replaced by process firewall rules. These results show that it is practical for the operating system to protect processes by preventing a variety of resource access attacks system-wide.

"Process Firewalls: Protecting Processes During Resource Access," Hayawardh Vijayakumar, Joshua Schiffman, Trent Jaeger. In Proceedings of the 2013 ACM European Conference on Computer Systems (EuroSys), April 2013.

We then explored the policy model underlying the process firewall approach. This resultant policy model highlighted two contributions: (1) the explicit definition of adversary models as adversarial roles, which list the permissions that dictate whether one subject is an adversary of another, and (2) the application of data-flow to determine the adversary control of the names used to retrieve resources. An evaluation using multiple adversary models shows that data-flow is necessary to authorize resource retrieval in over 90% of the system call instances. By making adversary models and the adversary accessibility of all aspects of resource retrieval explicit, we can block resource access attacks system-wide.

"Policy Models to Protect Resource Retrieval," Hayawardh Vijayakumar, Xinyang Ge, Trent Jaeger. In Proceedings of the 19th ACM Symposium on Access Control Models and Technologies (SACMAT), June 2014.

A downside of the previous work is that data-flow can be difficult to compute accurately in a non-type-safe

language, such as C. As a result, we then explored the principles that determine when a resource retrieval is unsafe. Critically, we developed *a comprehensive defense against vulnerabilities during resource retrieval* in this paper. First, we identify that the fundamental reason that resource retrieval vulnerabilities exist is *a mismatch between programmer expectations and the actual environment the program runs in*. To address such mismatches, we propose JIGSAW, a system that can automatically derive programmer expecta- tions and enforce it on the deployment. JIGSAW constructs programmer expectations as a name flow graph, which represents the data flows from the inputs used to construct file pathnames to the retrieval of system resources using those pathnames. We find that whether a program makes any attempt to filter such flows implies expectations about the threats the programmer expects during resource retrieval, the enabling JIGSAW to enforce those expectations. We evaluated JIGSAW on widely-used programs and found that programmers have many implicit expectations. These mismatches led us to discover two previously-unknown vulnerabilities and a default misconfiguration in the Apache webserver. JIGSAW enforces program expectations for approximately 5% overhead for Apache webservers, thus eliminating vulnerabilities during resource retrieval efficiently and in a principled manner.

"JIGSAW: Protecting resource access by inferring programmer expectations," Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, Trent Jaeger. In Proceedings of the 23rd USENIX Security Symposium, August 2014.

These results in this section culminated in the Ph.D. thesis of Hayawardh Vijayakumar, an advisee of Dr. Trent Jaeger. Hayawardh now works at Samsung Research America on the security-critical Knox project, which examines use of the ARM TrustZone architecture to improve the security of cellphone systems.

"Protecting Programs During Resource Access," Hayawardh Vijayakumar, Ph.D. dissertation, Penn State University, May 2014.

2.2 Kernel Software Execution Integrity

Computing systems now may utilize many types of privileged software, such as hypervisors, microkernels and their user-space servers, and/or conventional kernels. We will refer to this software collectively as *kernel software*. Traditionally, we trust kernel programmers to write code that protects the integrity of their kernel software during its execution. As a result, kernel software lacks mechanisms to protect its integrity during execution. However, we are now finding instances of malware that leverages kernel vulnerabilities to "root" system (rather than compromising provileged user-space processes), so protecting the integrity of kernel software during its execution is now becoming necessary. In this project, we explored the enforcement of two types of integrity properties: (1) restricting kernel software to only execute approved code and (2) restricting kernel software to fine-grained control-flow integrity.

First, current smartphone processors have hardware support for running a protected environment, such as the ARM TrustZone extensions, but such hardware does not ensure that the smartphone operating sytems only run approved code. In particular, a conventional operating system running with TrustZone still retains full control of memory management, which a rootkit can use to reconfigure memory management to circumvent W-xor-X protections, enabling adversaries to modify kernel code or execute data. We develop a novel mechanism called SPROBES that enables introspection of operating systems running on ARM TrustZone hardware, which can be used to mediate operations that could impact the integrity of memory management. Using SPROBES, an introspection mechanism protected by TrustZone can instrument individual operating system instructions of its choice, receiving an unforgeable trap whenever any SPROBE is executed. To protect memory management (and the SPROBES themselves), we identify a set of five invariants whose enforcement is sufficient to restrict rootkits to execute only approved, SPROBE-injected kernel code. We implemented a proof-of-concept version of SPROBES for the ARM Fast Models emulator, demonstrating that in Linux kernel 2.6.38, only 12 SPROBES are sufficient to enforce all five of these invariants. With SPROBES we show that it is possible to leverage the limited TrustZone extensions to limit conventional kernel execution to approved code comprehensively. This work appeared in the Mobile Security Technologies Workshop affiliated with the IEEE Symposium on Security and Privacy.

"SProbes: Enforcing Kernel Code Integrity on the TrustZone Architecture," Xinyang Ge, Hayawardh

Vijayakumar, Trent Jaeger. In Proceedings of the Mobile Security Technologies 2014 Workshop (MoST14), in conjunction with the IEEE Symposium on Security and Privacy), May 2014.

Second, even with restrictions to limit privileged kernel software to execute only approved code, code reuse attacks are still possible. To prevent code reuse attacks, researchers have recommended enforcing *control-flow integrity* (CFI). However, researchers have found it difficult to produce fine-grained control flow graphs (CFGs) to restrict adversaries and have expressed concerns about enforcing CFI efficiently. However, we have found that it is not only possible to compute a fine-grained CFG for kernel software, but we can leverage the fine CFG to enforce CFI more efficiently than coarse-grained CFI. To compute a fine-grained CFG, we find that kernel software programmers largely use function pointers in a restricted way that enables us to compute an accurate CFG using a static taint analysis. To enforce this CFG efficiently, we select optimal instrumentation for each indirect call/return site. We also leverage the fact that many indirect control transfers only have one target in kernel software.

We evaluate the effectiveness of the proposed fine-grained CFI mechanism to kernel software and apply the mechanism comprehensively to FreeBSD, the MINIX microkernel system, and MINIXs user-space servers, on Intel x86 platforms. We show that our approach eliminates over 70% of the indirect targets that are otherwise allowed by current fine-grained CFI techniques, while our implementation incurs 1.82% performance overhead on FreeBSD and 0.76% on MINIX on macrobenchmarks, and 11.91%/42.03% (average/maximum) and 2.02%/5.64% overhead on microbenchmarks, respectively, which are less overheads than a comparable coarse-grained CFI implementation. As a result, we find that fine-grained CFI can be practical and efficient for critical kernel software.

"Fine-Grained Control-Flow Integrity for Kernel Software," Xinyang Ge, Mathias Payer, Trent Jaeger. Penn State Institute of Networking and Security Research Technical Report, NAS-TR-0183-2015, May 2015.

3 Integrity Safety in Middleware

We also focused on the integrity of the use of extensible Web browsers. Most browsers support an extensible architecture, in which code written by untrusted third parties enhances the core functionality of the browser. This code is privileged, and can access most of the sensitive state stored in the browser, such as cookies, browsing history, and the contents of individual web pages loaded by the browser. Extension code is not sandboxed in the same way as web application code, using the same-origin policy.

To address such threats, modern Web browsers have actively sought to apply security principles to the design of extension code. The Mozilla Jetpack project and the Chrome extensions project aim to build a programming interface and the supporting infrastructure that will allow better control over the execution of untrusted third-party extension code. In particular, the projects aim to enforce the principles of least privilege and privilege separation, and attempt to compartmentalize extension code so as to limit the impact of exploits against vulnerable extensions.

This part of the project had two main goals. The first goal was to evaluate the extent to which modern extension architectures achieve their security goals. Specifically, we used static analysis to study capability leaks in Jetpack modules and add-ons, i.e., cases where code violates modularity by leaking a pointer to a privileged resource to another module. We implemented Beacon, a static analysis tool to identify the leaks and used it to analyze 77 core modules from the Jetpack framework and another 359 Jetpack add-ons. In total, Beacon analyzed over 600 Jetpack modules and detected 12 capability leaks in 4 core modules and another 24 capability leaks in 7 Jetpack add-ons. Beacon also detected 10 over-privileged core modules. We shared the details with Mozilla's development team, who have acknowledged our findings. These results were published in our paper at ECOOP'12:

"An Analysis of the Mozilla Jetpack Extension Framework," Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, Chung-chieh Shan, Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP 2012). Published as Volume 7313 of Lecture Notes in Computer Science (LNCS), pages 333-355; Beijing, China; June 11-16, 2012.

Having characterized the security of extension frameworks, our second goal was to tool to systematically port legacy browser extensions to these modern frameworks. Specifically, we built Morpheus, a tool that retargets legacy

extensions for the Mozilla Firefox browser, and ports them to the Jetpack framework. Morpheus uses static Javascript analysis and transformation to automatically compartmentalize legacy browser extensions and make them modular. In our experimental evaluation, we have applied Morpheus to port 52 legacy Firefox extensions to the Jetpack framework. The results of this work were published in our paper at ECOOP'14:

"Retargetting Legacy Browser Extensions to Modern Extension Frameworks," Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP 2014). Published as Volume 8586 of Lecture Notes in Computer Science (LNCS), pages 463-488; Uppasala, Sweden; July 28-August 1, 2014.

4 Integrity Safety in Programs

Researchers have shown that the addition of security mechanisms can have significant positive impact on protecting the integrity of programs and the data that they process. For example, reference validation mechanisms that satisfy the *reference monitor concept* can ensure that the program protects data integrity. The challenge is for programmers to extend their programs with such mechanisms correctly. However, adding such mechanisms manually is a complex and error-prone task, taking several years and leading to new vulnerabilities. In this part of the project, we have explored mostly-automated mechanisms to retrofit legacy programs with security code to protect data integrity.

First, several recent operating systems provide system calls that allow an application to explicitly manage the privileges of modules with which the application interacts. Such privilege-aware operating systems allow a programmer to a write a program that satisfies a strong security policy, even when it interacts with untrusted modules. However, it is often non-trivial to rewrite a program to correctly use the system calls to satisfy a high-level security policy. This paper concerns the policy-weaving problem, which is to take as input a program, a desired high-level policy for the program, and a description of how system calls affect privilege, and automatically rewrite the program to invoke the system calls so that it satisfies the policy. We present an algorithm that solves the policy-weaving problem by reducing it to finding a winning modular strategy to a visibly pushdown safety game, and applies a novel game-solving algorithm to the resulting game. Our experiments demonstrate that our algorithm can efficiently rewrite practical programs for a practical privilege-aware system.

"Secure Programming via Visibly Pushdown Safety Games," William R. Harris, Somesh Jha, Thomas W. Reps. In Proceedings of Computer Aided Verification (CAV), pgs. 581-598, July 2012.

Second, we explored a mostly-automated approach to augment servers that manage resources on behalf of multiple, mutually-distrusting clients to mediate access to those resources to ensure that each client request complies with an authorization policy. This goal is typically achieved by placing authorization hooks at appropriate locations in server code. The goal of authorization hook placement is to completely mediate all security-sensitive operations on shared resources. We proposed an automated hook placement approach that is motivated by a novel observation *that the deliberate choices made by clients for objects from server collections and for processing those objects must all be authorized*. We have built a tool that uses this observation to statically analyze the server source. Using real-world examples (the X server and postgresql), we show that the hooks placed by our method are just as effective as hooks that were manually placed over the course of years while greatly reducing the burden on programmers.

"Leveraging 'Choice' to Automate Authorization Hook Placement," Divya Muthukumaran, Trent Jaeger, Vinod Ganapathy. In Proceedings of ACM Conference on Computer and Communications Security (CCS), October 2012.

We developed a mostly-automated method to transform a set of commodity MAC policies in hosts, programs, and virtualization platforms into a system-wide policy that proactively protects system integrity, approximating the Clark-Wilson integrity model. The method uses the insights from the Clark-Wilson model, which requires integrity verification of security-critical data and mediation at program entrypoints, to extend existing MAC policies with the proactive mediation necessary to protect system integrity. We demonstrated the practicality of producing Clark-Wilson policies for distributed systems on a web application running on virtualized Ubuntu SELinux hosts,

where our method finds: (1) that only 27 additional entrypoint mediators are sufficient to mediate the threats of remote adversaries over the entire distributed system and (2) and only 20 additional local threats require mediation to approximate Clark-Wilson integrity comprehensively. As a result, available security policies can be used as a foundation for proactive integrity protection from both local and remote threats.

"Transforming Commodity Security Policies to Enforce Clark-Wilson Integrity," Divya Muthukumaran, Sandra Rueda, Nirupama Talele, Hayawardh Vijayakumar, Trent Jaeger, Jason Teutsch, Nigel Edwards. In Proceedings of Annual Computer Security Applications Conference (ACSAC), December 2012.

Next, we explored methods to retrofit programs for specific security enforcement systems, such as the Capsicum capability system, allowing a programmer to write an application that satisfies strong security properties by invoking security-specific system calls at a few key points in the program. However, rewriting an application to invoke such system calls correctly is an error-prone process: even the Capsicum developers have reported difficulties in rewriting programs to correctly invoke system calls. We developed a system called capweave, a tool that takes as input (i) an LLVM program, and (ii) a declarative policy of the possibly-changing capabilities that a program must hold during its execution, and rewrites the program to use Capsicum system calls to enforce the policy. Our experiments demonstrate that capweave can be applied to rewrite security-critical UNIX utilities to satisfy practical security policies. capweave itself works quickly, and the runtime overhead incurred in the programs that capweave produces is generally low for practical workloads.

"Declarative, Temporal, and Practical Programming with Capabilities," William R. Harris, Somesh Jha, Thomas W. Reps, Jonathan Anderson, Robert N. M. Watson. In Proceedings of the IEEE Symposium on Security and Privacy, pgs. 18-32, May 2013.

Next, we explored the development of an interactive program analysis that programmers can apply to validate that their manual optimizations do not change his programs semantics. Our analysis casts the problem of validating an optimization as an abductive inference problem in the context of checking program equivalence. Our analysis solves the abductive equivalence problem by interacting with the programmer so that the programmer implements a solver for a logical theory that models library functions invoked by the program. We have used our analysis to validate optimizations of real-world, mature applications: the Apache software suite, the Mozilla Suite, and the MySQL database.

"Validating Library Usage Interactively," William R. Harris, Guoliang Jin, Shan Lu, Somesh Jha. In Proceedings of Computer Aided Verification (CAV), pgs. 796-812, July 2013.

We also published an abstract summarizing the approach of game-based synthesis for the development of programs that enforce integrity requirements.

"Secure programs via game-based synthesis," Somesh Jha, Thomas W. Reps, William R. Harris. In Proceedings of Formal Methods in Computer-Aided Design, pgs. 12-13, October 2013.

Lastly, we explored algorithms that automatically compute a minimal authorization hook placement that satisfies constraints that describe desirable access control policies. These *authorization constraints* describe expectations about the access control policies that a program will enforce. Such constraints reduce the space of enforceable access control policies (i.e., those policies that can be enforced given a hook placement that satisfies the constraints), but enable significant reduction in the number of authorization hooks required to enforce policies that satisfy the constraints as well. We have built a tool that implements this authorization hook placement method, demonstrating how programmers can produce authorization hooks for real-world programs and leverage policy goal-specific constraint selectors to automatically identify many authorization constraints. Our experiments show that our technique reduces manual programmer effort by as much as 58% and produces placements that reduce the amount of policy specification by as much as 30%.

"Producing Hook Placements To Enforce Expected Access Control Policies," Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. In Proceedings of Engineering Security Software and Systems

(ESSoS), March 2015.

These results in this section culminated in the Ph.D. theses of Divya Muthukumaran, an advisee of Dr. Trent Jaeger, and William Harris, an advisee of Dr. Somesh Jha. Dr. Muthukumaran is now a post-doctoral scholar at Imperial College London, and Dr. Harris is an assistant professor at Georgia Tech.

"Automating Authorization Hook Placement in Programs," Divya Muthukumaran, Ph.D. dissertation, Penn State University, August 2013.

"Secure Programming via Game-based Synthesis", William Harris, Ph.D. dissertation, University of Wisconsin, Madison, December 2014.

1.

1. Report Type

Final Report

Primary Contact E-mail
Contact email if there is a problem with the report.

tjaeger@cse.psu.edu

Primary Contact Phone Number
Contact phone number if there is a problem with the report

814-865-1042

Organization / Institution name

Penn State University

Grant/Contract Title
The full title of the funded effort.

Information Flow Integrity for Systems of Independently-Developed Components

Grant/Contract Number
AFOSR assigned control number. It must begin with "FA9550" or "F49620" or "FA2386".

FA9550-12-1-0166

Principal Investigator Name
The full name of the principal investigator on the grant or contract.

Trent Jaeger

Program Manager
The AFOSR Program Manager currently assigned to the award

Tristan Nguyen

Reporting Period Start Date

04/01/2012

Reporting Period End Date

03/31/2015

Abstract

The aim of this project is to enable enforcement of integrity safe in systems of independently-developed components. In this project, we explore this problem from three perspectives. First, we developed integrity safety properties and mechanisms to enforce them. In particular, we developed resource retrieval (access) integrity, which protects programs when retrieving system resources, and implemented the process firewall mechanism to enforce this property. Second, we developed integrity safety mechanisms for a variety of software, including web browsers (to protect them from browser extensions), kernel software (to enforce resource retrieval integrity and fine-grained control-flow integrity of approved code), and user-space programs (to enforce access control policies). Third, we developed methods to retrofit software to enforce integrity safety properties mostly-automatically through safety games and authorization constraints. Both of these methods enable an efficient deployment of code to enforce expected integrity requirements. This work has been

published several top conferences in computer security and programming languages and some of the projects have been packaged for open-source distribution.

Distribution Statement

This is block 12 on the SF298 form.

Distribution A - Approved for Public Release

Explanation for Distribution Statement

If this is not approved for public release, please provide a short explanation. E.g., contains proprietary information.

SF298 Form

Please attach your SF298 form. A blank SF298 can be found here. Please do not password protect or secure the PDF The maximum file size for an SF298 is 50MB.

AFD-070820-035.pdf

Upload the Report Document. File must be a PDF. Please do not password protect or secure the PDF . The maximum file size for the Report Document is 50MB.

final.pdf

Upload a Report Document, if any. The maximum file size for the Report Document is 50MB.

Archival Publications (published) during reporting period:

”Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies,” Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, Trent Jaeger. In Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS), May 2012.

”STING: Finding Name Resolution Vulnerabilities in Programs,” Hayawardh Vijayakumar, Joshua Schiffman, Trent Jaeger. In Proceedings of the 21st USENIX Security Symposium, August 2012.

”The right files and the right time,” Hayawardh Vijayakumar and Trent Jaeger. In Proceedings of the 5th Symposium on Configuration Analytics and Automation (SafeConfig), October 2012.

”Process Firewalls: Protecting Processes During Resource Access,” Hayawardh Vijayakumar, Joshua Schiffman, Trent Jaeger. In Proceedings of the 2013 ACM European Conference on Computer Systems (Eu- roSys), April 2013.

”Policy Models to Protect Resource Retrieval,” Hayawardh Vijayakumar, Xinyang Ge, Trent Jaeger. In Proceedings of the 19th ACM Symposium on Access Control Models and Technologies (SACMAT), June 2014.

”JIGSAW: Protecting resource access by inferring programmer expectations,” Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, Trent Jaeger. In Proceedings of the 23rd USENIX Security Symposium, August 2014.

”Protecting Programs During Resource Access,” Hayawardh Vijayakumar, Ph.D. dissertation, Penn State University, May 2014.

”SProbes: Enforcing Kernel Code Integrity on the TrustZone Architecture,” Xinyang Ge, Hayawardh Vijayakumar, Trent Jaeger. In Proceedings of the Mobile Security Technologies 2014 Workshop (MoST14), in conjunction with the IEEE Symposium on Security and Privacy), May 2014.

”Fine-Grained Control-Flow Integrity for Kernel Software,” Xinyang Ge, Mathias Payer, Trent Jaeger. Penn State Institute of Networking and Security Research Technical Report, NAS-TR-0183-2015, May 2015.

”An Analysis of the Mozilla Jetpack Extension Framework,” Rezwana Karim, Mohan Dhawan, Vinod

Ganapathy, Chung-chieh Shan, Proceedings of the 26th European Conference on Object-Oriented Program- ming (ECOOP 2012). Published as Volume 7313 of Lecture Notes in Computer Science (LNCS), pages 333- 355; Beijing, China; June 11-16, 2012.

"Retargetting Legacy Browser Extensions to Modern Extension Frameworks," Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP 2014). Published as Volume 8586 of Lecture Notes in Computer Science (LNCS), pages 463-488; Uppasala, Sweden; July 28-August 1, 2014.

"Secure Programming via Visibly Pushdown Safety Games," William R. Harris, Somesh Jha, Thomas W. Reps. In Proceedings of Computer Aided Verification (CAV), pgs. 581-598, July 2012.

"Leveraging 'Choice' to Automate Authorization Hook Placement," Divya Muthukumaran, Trent Jaeger, Vinod Ganapathy. In Proceedings of ACM Conference on Computer and Communications Security (CCS), October 2012.

"Transforming Commodity Security Policies to Enforce Clark-Wilson Integrity," Divya Muthukumaran, Sandra Rueda, Nirupama Talele, Hayawardh Vijayakumar, Trent Jaeger, Jason Teutsch, Nigel Edwards. In Proceedings of Annual Computer Security Applications Conference (ACSAC), December 2012.

"Declarative, Temporal, and Practical Programming with Capabilities," William R. Harris, Somesh Jha, Thomas W. Reps, Jonathan Anderson, Robert N. M. Watson. In Proceedings of the IEEE Symposium on Security and Privacy, pgs. 18-32, May 2013.

"Validating Library Usage Interactively," William R. Harris, Guoliang Jin, Shan Lu, Somesh Jha. In Proceedings of Computer Aided Verification (CAV), pgs. 796-812, July 2013.

"Secure programs via game-based synthesis," Somesh Jha, Thomas W. Reps, William R. Harris. In Proceedings of Formal Methods in Computer-Aided Design, pgs. 12-13, October 2013.

"Producing Hook Placements To Enforce Expected Access Control Policies," Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. In Proceedings of Engineering Security Software and Systems (ESSoS), March 2015.

"Automating Authorization Hook Placement in Programs," Divya Muthukumaran, Ph.D. dissertation, Penn State University, August 2013.

"Secure Programming via Game-based Synthesis", William Harris, Ph.D. dissertation, University of Wisconsin, Madison, December 2014.

Changes in research objectives (if any):

None

Change in AFOSR Program Manager, if any:

From Robert Herklotz to Tristan Nguyen

Extensions granted or milestones slipped, if any:

None

AFOSR LRIR Number

LRIR Title

Reporting Period

Laboratory Task Manager

Program Officer

Research Objectives

Technical Summary

Funding Summary by Cost Category (by FY, $K)

	Starting FY	FY+1	FY+2
Salary			
Equipment/Facilities			
Supplies			
Total			

Report Document

Report Document - Text Analysis

Report Document - Text Analysis

Appendix Documents

2. Thank You

E-mail user

May 30, 2015 10:52:24 Success: Email Sent to: tjaeger@cse.psu.edu