Reinventing the Privilege Drop

How Principled Preservation of Programmer Intent Would Prevent Security Bugs

Ira Ray Jenkins
jenkins@cs.dartmouth.edu
Dartmouth College

Sergey Bratus
sergey@cs.dartmouth.edu
Dartmouth College

Sean Smith
sws@cs.dartmouth.edu
Dartmouth College

Maxwell Koo
Narf Industries
maxk@narfindustries.com

ABSTRACT

The principle of least privilege requires that components of a program have access to only those resources necessary for their proper function. Defining proper function is a difficult task. Existing methods of privilege separation, like Control Flow Integrity and Software Fault Isolation, attempt to infer proper function by bridging the gaps between language abstractions and hardware capabilities. However, it is programmer intent that defines proper function, as the programmer writes the code that becomes law. Codifying programmer intent into policy is a promising way to capture proper function; however, often onerous policy creation can unnecessarily delay development and adoption.

In this paper, we demonstrate the use of our ELF-based access control (ELFbac), a novel technique for policy definition and enforcement. ELFbac leverages the common programmer's existing mental model of scope, and allows for policy definition at the Application Binary Interface (ABI) level. We consider the roaming vulnerability found in OpenSSH, and demonstrate how using ELFbac would have provided strong mitigation with minimal program modification. This serves to illustrate the effectiveness of ELFbac as a means of privilege separation in further applications, and the intuitive, yet robust nature of our general approach to policy creation.

CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation*; *Software security engineering*;

KEYWORDS

OpenSSH, LangSec, ELFbac, vulnerability mitigation, privilege separation

ACM Reference Format:
Ira Ray Jenkins, Sergey Bratus, Sean Smith, and Maxwell Koo. 2018. Reinventing the Privilege Drop: How Principled Preservation of Programmer Intent Would Prevent Security Bugs. In *HoTSoS '18: Hot Topics in the Science of Security: Symposium and Bootcamp, April 10–11, 2018, Raleigh, NC, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3190619.3190635

1 INTRODUCTION

The principle of least privilege states that components of a system should use the least set of privileges necessary for correct operation [18]. The goal of this concept is to limit the exposure of vulnerabilities within a system. A non-administrative user should have more restrictive access control permissions than a system administrator. A web browser should generally not have default access to critical operating system files. In this way, ideally, a vulnerability in one component (e.g., a web browser) should have no impact on the overall operation of the system.

More than twenty-five years after Saltzer and Schroeder's work, in 2003, Provos, Friedl, and Honeyman recognized that least privilege should be applied not just to the interprocess workings of a system (e.g., those between programs), but also to the interactions amongst individual units of execution within a single program (e.g., functions) [15]. Their idea was to divide system services (e.g., network daemons) into pre- and post-authentication phases, with functionality and access control permissions restricted according to the context of each phase.

In the last decade, the authors introduced Executable and Linkable Format (ELF) based access control (ELFbac) [5, 6], a direct successor to privilege separation as applied to intraprocess memory isolation. ELFbac is a technique for constructing security policies at the Application Binary Interface (ABI) level. ABIs establish the rules of interaction between pre-compiled code objects, such as functions, libraries and operating systems. These rules include definitions for primitive data types, calling conventions, and the binary formatting of said objects. This is a natural place to denote privilege separation because the semantics of scope and privilege already exist.

In this paper, we present a case study of ELFbac's use in mitigating CVE-2016-0777, the "roaming bug" found in the OpenSSH client. Through this vulnerability, a malicious server can fool a client into disclosing private data, including authentication keys. ELFbac allows us to prevent this potential disclosure of client-side private key material, and in general, to enforce the principle of least privilege over multiple program execution units with existing memory isolation mechanisms. Utilizing ABI-level policy enforcement, we recapture the privilege separation inherent in a programmer's mental model of scope, and thus reinvent the privilege drop at a lower level than seen before.

HoTSoS '18, April 10–11, 2018, Raleigh, NC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6455-3/18/04…$15.00
https://doi.org/10.1145/3190619.3190635

The rest of this paper is organized as follows: Section 2 justifies our choice of OpenSSH as a target, and presents our view of policy in the role of vulnerability mitigation. Section 3 reviews the vulnerability found in OpenSSH. Section 4 re-introduces the reader to ELFbac. In Section 5, we describe an ELFbac policy that isolates the unintended access patterns that enabled the roaming bug. We generalize a methodology for policy creation in Section 6. Finally, we present our concluding thoughts in Section 7.

2 BACKGROUND

In this section, we provide background on OpenSSH, policy mitigations, and the role of intent in policy mechanisms.

Picking on OpenSSH

The Secure Shell (SSH) network protocols are the most popular method of securing remote communications over insecure networks. The ubiquity of SSH is demonstrated by its being ported to nearly all modern hardware and operating systems. The popularity of SSH is such that adversarial scanning for SSH installations with default credentials is continual on the internet. Any connected host, unless protected by a firewall, can expect at least dozens, if not hundreds, of scans in an hour. SSH is the primary means of securely managing internet routers (e.g., Cisco and Juniper) and many other devices that offer command line shell interfaces. In a nutshell, SSH is a key part of the management infrastructure for both the internet and the Internet of Things (IoT).

OpenSSH is a free and open-source implementation of the SSH protocols. Since its release in 1999, OpenSSH has been the most popular implementation for securing remote communications. OpenSSH is the de-facto standard for encrypted remote communications on Unix-like systems. Its usefulness is evidenced by the continual addition of new features, such as connection and credential forwarding, fully functional virtual private networks (VPNs) and a variety of methods for constructing fully functional VPNs. SSH is the most light-weight method for remote access to embedded devices that provides cryptographic protections without the full the weight of a public key infrastructure (PKI).

The OpenSSH codebase is regarded to be one of the most dependable of its kind for security. Not surprisingly, announcements of OpenSSH vulnerabilities are critical security news. These vulnerabilities are not frequent, owing to OpenSSH's simple and principled design and architecture. But their impact is so large that every time they happen, they inspire not just fixes, but new security mitigation mechanisms. Discovering pre-authentication vulnerabilities in OpenSSH has been enough for an attacker to achieve instant notoriety (e.g., as with 2002 GOBBLES remote code execution bug [2]). The discovery of GOBBLES, a pre-authentication, challenge-response vulnerability, underscored the point that input parsing vulnerabilities are not a thing of the past, even in highly audited and concise codebases, such as OpenSSH.

OpenSSH was the original target for Provos, Friedl, and Honeyman's privilege separation [15]. Their rearchitected design of OpenSSH moved the handling of credentials into a separate restricted environment. Each new remote connection spawns a new process (referred to as the "master") that must run in a privileged mode in order to allow authentication of key exchanges and the creation of further processes with the privileges of an authenticated user. With privilege separation, instead of handling a new connection on its own (within the context of its privileged mode), the master process forks "slave" processes without privileges [1] to handle new connections. These slave processes do not have the access control permissions to create pseudo-terminals, authenticate keys, or create new processes. Instead, they may only request these actions be performed on their behalf by the master process. In this manner, vulnerabilities exploited within a slave process should be isolated by its unprivileged execution mode.

Thus, the discovery in 2016 of a new remote vulnerability in OpenSSH, the "roaming bug" [4], has been big news in the security world, and it was one of the vulnerabilities that got its own themed coverage in industry press [20]. As before, the weakness behind this vulnerability was due to parsing of attacker crafted data. Previous mitigations for such parsing flaws have failed. Since there is no generic way of eliminating parsing vulnerabilities in C/C++ code, a new class of mitigation is in order. We present a policy and the mitigation of this kind.

Our choice of OpenSSH as the target for applying our policy mitigation is not accidental. As a direct successor to privilege separation, ELFbac can be shown to easily mitigate certain classes of vulnerabilities in a wide variety of applications, including OpenSSH. Additionally, as in several prior instances, OpenSSH—due to its infrastructure importance and exposure—is perhaps the best target for showcasing a new mitigation.

Vulnerability Mitigation with Policy

Many vulnerabilities arise as a result of a mismatch between a programmer's mental model of software and the reality that exists when computation is performed in real-world environments. Software exploits, by definition, induce unintended computation; thereby becoming proofs-by-construction (in a practical and mathematical sense) to the discrepancies present between a programmer's intent and a program's actual behavior. Were programmer intent explicitly codified in policy, policy enforcement mechanisms would become the de facto gatekeeper between intended computation and exploitation.

Unfortunately, preceding policy approaches have fallen short in protecting user space code from network inputs. The standard process model lets all code touch all data in the address space. Policy inference methods that rely on language features or hardware capabilities ignore the programmer's intent. In contrast, ELFbac allows programmers to specify the intent of semantically distinct, intra-process relationships (found between code and data) at the granularity of ELF sections (the primary container of code and data within ELF), and to separate program components (such as functions, modules, and libraries) within a process' address space. This delineation of code and data relationships aligns naturally with the common programmer's understanding of scoping.

Policy definition and implementation must be simple and straightforward to be useful. Policies must strike a balance between the simplicity of creation, and the aggressiveness of enforcement. Prior

[1] In many Unix variants, the privileges of a child process are inherited from its parent. These privileges must be abandoned by the child, effectively leading to the phrase "privilege drop."

policy mechanisms have shown that, in practice, policies can be overly pedantic and unwieldy, forcing great cognitive burden on the policy creator. Alternatively, policies may be overly broad, and thus, frustrate users, resulting in the circumvention or outright abandonment of policy enforcement. Policy definition within ELFbac is less onerous than previous methods. Policies are defined in a familiar language, and can be tailored to specific code and data, whether functions, whole libraries, or entire applications. In this way, policy creation may be limited to only those parts of an application that are security critical. Meanwhile, policy enforcement still allows for the broad capture of unintended computation. ELFbac policy enforcement monitors the defined code and data units and only allows explicitly defined interaction patterns. Enforcement is accomplished through existing virtual memory management code found within the kernel. Undesirable computation becomes illegal memory access, resulting in a segmentation fault. This approach to policy definition leverages a programmer's own understanding of security boundaries within a program, and enforcement renders common classes of vulnerabilities inert by disallowing unspecified computation and data disclosure.

Reinventing the Privilege Drop

We regard our policy approach as a continuation of the classic privilege separation design, which is an instance of the least privilege principle. The classic privilege drop policy primitive postulates that an application should signal the system when it no longer needs a privilege it had to start with, so that any subsequent attempts by the application code to perform the privileged operation are flagged as policy violations and invoke a policy response, such as killing the process. In network daemons, listening for connections on a non-emphemeral port and spawning a user's sessions required root privilege; whereas, subsequent processing of the user's data typically did not. Thus, dropping privileges expressed the programmer intent to no longer use root privileges after a particular point in time.

Some suggestions extended this further—for example, DJ Bernstein's *disablenetwork()* system call [7] would signal the OS that a program intended to initiate no new network operations from this point on. Implementing some form of privilege drop has become an expected design element for network daemons, proving to be an effective mitigation. The idea of dropping privileges similarly underlies the design of SELinux. SELinux mediates system calls based on the security label of a process and the security label of an object, such as a file or socket, involved in Linux system calls. As processes get spawned by their respective parents (that chain of parents ultimately going up to init), every *execv()* system call checks the policy rules and assigns a new security domain to the new process. These security domains restrict the access rights of the child process to just the ones specified by policy. Notably, SELinux treats the access permissions it enforces as a "bag of permissions" that a process, once created with a particular security label, can exercise in any order and any number of times.

Previous policy systems focused on access by processes to a system's objects, such as files. And aimed to capture the programmer intent with regard to such accesses. Although classic and still indispensable, this view of intents is insufficient for modern applications which may keep their objects in memory and never really trigger a disc operation (which is mitigated by the policy mechanism) until it is too late. For example, the attacker may be after a cached copy of a cryptographic key in a daemon's memory, or after the integrity of a memory representation of an object describing access permissions. Thus, in the presence of ubiquitous caching and other performance optimizations, the policy must also mitigate access to such objects to be effective. SE Linux and other mandatory access control systems relying on file accesses have become less effective in a world where not everything you care about is neatly encapsulated in a file; contrary to the Unix guiding principle that everything is a file.

This brings us to the necessity of defining a new class of programmer intent, namely intraprocess memory references by units of execution within a program. As a modern program gets assembled out of many functions, classes, modules, and libraries, the programmer intent with respect to these components' ability to access certain kinds of data becomes important. For example, the programmer likely does not intend for an image processing library used by a web-server to access the server's certificates. Yet this is exactly what happened in a number of exploits wherein a crafted image triggered the image processing library vulnerability to corrupt data units in memory unrelated to image processing [3].

Thus, a programmer may be interested in protecting his code from unintended interactions with the library code that his program would not use, but that is still loaded as a part of a DLL or shared object. Conversely, a library author maybe interested in protecting the integrity of the data his library handles from the code that handles host-style inputs, and may be exploited by them. Existing policy solutions provide no support for expressing such intents, other than performing the library call into a dedicated, newly spawned process. This solution is highly effective [8], but too expensive for modern multithreaded servers.

Programmer intents and expectations with respect to libraries can be generalized to any code units, such as those contained in a C-compilation unit (where e.g, shared objects may be declared "file-scoped" to underscore that they are not supposed to be referenced, and therefore accessed outside of the file by any code outside the given file). Indeed, such intent idioms are broadly used in the Linux kernel itself.

This Paper

This case study focuses on a particular kind of intraprocess access intent which is fundamental for network daemons and any other code that must process untrusted data. The logic of input validation implies that potentially host-style crafted data passes through a validation code-unit after which it is assumed validated and safe for the rest of the code to process. We regard the ability of code to access raw input as a dangerous privilege that must be constrained to only the code that needs it. This insight comes from experience of exploitation of parser bugs and in the very code that was supposed to validate inputs, as was the case with several high profile OpenSSH vulnerabilities, including the roaming bug. Conversely, the code that validates input data is not intended to access other kinds of data generated during the processing of inputs. Our policy system allows the programmer to specify these expectations at the level of code units, and to have them enforced by the kernel. In this sense, the policy mechanism we present is a direct successor of the privilege drop applied to memory access.

3 ROAMING IN OPENSSH

OpenSSH is a free implementation of Secure Shell (SSH), a suite of programs widely used to secure remote communications over unsecured networks. As noted earlier, we consider it here because of its open-source, modular architecture, and ubiquitous presence on modern operating systems. Of particular interest to the authors is its use within Industrial Control Systems (ICS) and the IoT. With the growing prevalence of off-the-shelf hardware and commercial software being utilized within industry, in conjunction with the popularity of OpenSSH's (often default) deployment and the reality of recent vulnerabilities targeting it, the security of remote communications is increasingly important and precarious.

In version 5.4, released in 2010, the OpenSSH client introduced an experimental and undocumented "roaming" feature. The purpose of roaming was to allow the resumption of suspended sessions, e.g., in the case of unexpected network termination. To accomplish this feature, upon session interruption, the client would maintain a buffer of (unsent) messages to send to the server upon reconnect.

In 2016, CVE-2016-0777 disclosed an information leak present in the implementation of OpenSSH's roaming feature [4]. Although roaming was never officially supported by the OpenSSH server-side, a malicious or compromised server could persuade an OpenSSH client (installed with default settings) to send arbitrary data via the roaming protocol, including potentially exposing private SSH keys.

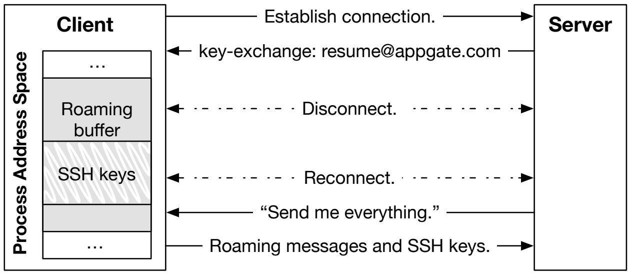

Figure 1: The information leak within OpenSSH. A malicious server convinces a client to return sensitive data upon a roaming reconnect.

Figure 1 shows a simplified execution leading to data exposure. Prior to an SSH session being established, connection parameters are negotiated between client and server. Presumably, the client will have previously loaded a user's private SSH keys into memory. Ideally, these keys would be expunged; however, in practice misuse of library functions (such as *fopen()*) and internal IO buffering allows the memory to persist. Part of the authentication protocol handshake permits a special key-exchange algorithm, "resume@appgate.com", sent by the server. A default client would see this identifier, and initiate the roaming protocol. After successfully establishing a connection and authenticating, a client-side buffer is created to store messages in the event of an interruption. As shown in the figure, the roaming buffer allocation overlaps the memory currently storing the (un-expunged) private SSH keys. During a disconnection, the client stores messages within this buffer. In the event of a reconnect, the server can request an arbitrary amount of the buffer to be re-sent, including parts of the buffer that were never written. To successfully achieve full exploitation, a server needs to perform some heap massaging to control the desired return-data, guess the client-side buffer size, and request all available data within that buffer. As a result, data that may have been previously free'd but not overwritten (for example, private SSH keys) that overlap the client-side buffer allocation are openly available to the server.

This exploit is not complicated, requiring no user interaction, and the risk of confidentiality exposure is quite high [14]. However, this vulnerability was given a medium ranking due to several mitigating factors, chief among them, the ease by which the roaming feature could be turned off and that server-side roaming had never been released. Therefore, the only vulnerability exposure was to malicious servers.

However, its existence highlights a key concept of secure programming, that of intent. An experimental feature was released publicly on the client-side, without corresponding code on the server-side. It is clear that, in the event of the feature's release, the developer(s) never intended for the server to control how much data was returned, or that the returned data might include confidential credentials. This is where security policy, and ELFbac in particular, become relevant.

Several mismorphisms [2] in concert could contribute to this vulnerability being exploited: use of a server controlled buffer size, use of previously allocated but uncleared memory (malloc vs. calloc), unknown dependence on internal IO buffering, and trusting the server to decide how much data is re-sent.

For a more detailed analysis, see the Qualys security advisory [16].

4 ELF-BASED ACCESS CONTROL

Modern programming is a miracle of abstraction. Complex programs are built through the composition of variables, functions, classes, modules, and libraries pulled from an increasingly complex hierarchy of code sources, both internal and external to a given project. Trustworthy computation is an intricate dance among the wildflowers of intent—those being planted by the programmer (under the design constraints of the current project), and those seeded by the creators of ABIs, compilers, and libraries used and borrowed from to create the final result.

Executable and Linkable Format (ELF) files contain the code and data for a given executable, as well as metadata necessary for the creation of a process address space. The operating system kernel utilizes the information within the ELF file to link, load, and ultimately construct a runtime process.

ELF files comprise sections and segments. Sections contain the code and data of a program, with as many 30 sections within a single executable. Each section defines semantically distinct units of code and data; with many having exclusive intersectional relationships, such as certain data readable or writable by only specific code. Many of the default relationships are predefined by language or runtime standards, such as glibc's initialization prior to *main*(),

[2] The authors kindly appropriate this term from the work of Smith, Koppel, Blythe, and Kothari on human policy and the reality of security circumvention. [19] We view software vulnerabilities as a natural extension of such differential perception.

code relocation, and dynamic linking. However, sections and their relationships are easily customizable at compile time with special compiler pragmas and linker scripts.

Sections are packed together by the linker to create segments. This packing is primarily a hold-over optimization based on limited memory, smaller caches, and perceived address space scarcity. *The ELF format naturally captures code and data semantics intended by the programmer—but the loader then discards this information!* For example, read-only data contained in non-executable sections can be grouped with default, executable code sections. This often violates the mental model and intent of the programmer. As a result, several “weird machines” [9], or unexpected computational engines, have been developed to take advantage of these mismorphisms via techniques such as memory corruption, control flow manipulation, and code reuse.

ELF sections are vehicles for programmer intent, and are thus natural security policy primitives. In this light, the linker becomes an expressive policy tool, useful for defining intersectional relationships between code and data. ELFbac attempts to reclaim the intended semantics of the programmer by leveraging an “unforgetful” loader to preserve the section identity of loaded memory segments. Thus, the loader becomes a policy enforcement mechanism, responsible for mapping sections into a process address space and setting up traps for unintended accesses within the virtual memory system.

Figure 2 shows the ELFbac architecture. The compilation phase of legacy code requires minimal, if any, modification. ELFbac policies are defined separately in a standard linker script format. An ELFbac-aware linker[3] then maps the legacy object code according to the defined policy. The final step prior to runtime is to create a process address space and load the now-combined objects into place. This is done with the help of ELFbac’s “unforgetful” loader, that does not discard the section metadata, but rather enforces the relationships specified. During runtime, a kernel shim utilizes the policy state machine to enforce access control.

ELFbac policies define a set of rules codifying the semantic relationships between data and code, specifically the access controls (i.e., read, write, and execute permissions) associated with their encapsulating sections. Policies are represented by finite state machines (FSM), with each state defining a particular abstract phase of program execution driven by a given section of code. State transitions are achieved via memory accesses (“data transitions”) and function calls (“call transitions”). Each transition rule specifies a source and a destination state and the interval of virtual addresses that trigger the transition. Throughout program execution, state integrity is maintained via the following invariants: (1) the program counter points to a location within the current state’s code section, and (2) control flow has proceeded from either the initial state or an allowed previous state.

Policy implementation relies on replacing the kernel’s view of a process’ virtual memory context with a diversified collection of “shadow” contexts, each representing a single policy state. Each shadow context only maps those regions of memory that can be accessed in the current state according to the policy. Permitted accesses transition the state machine, and may update the virtual memory context of the process. Meanwhile, any policy violations (unintended memory accesses or function calls) are trapped, leading to error handling code or ultimately a segmentation fault.

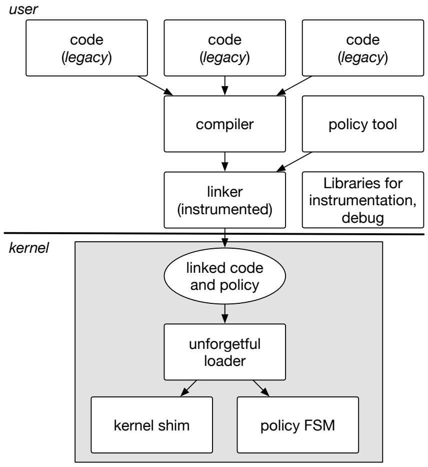

Figure 2: The ELFbac architecture. Legacy code is compiled and linked with an ELFbac policy. During runtime, an ELFbac-aware loader and kernel shim enforce the policy via FSM state transitions.

Utilizing ELFbac policy allows the enforcement of simple semantics, such as “input data can only be read by parsing routines” or “cryptographic keys should only be read or modified by cryptographic code.” In general, Turing-completeness makes it difficult to issue substantive statements about a program and its execution. In fact, the Rice-Shapiro theorem [17] shows that proving such sufficiently complex statements about a program is undecidable. However, as finite state machines, ELFbac policies are, at least computationally, much easier to reason about. For example, it is trivial to prove a policy that enforces “data from the filesystem must be encrypted before being sent over the network.” This could be achieved simply by isolating all network related code into a single state, and requiring all data transitions into that state to originate from a cryptographic state. In this way, ELFbac guarantees that all data will have been encrypted before being sent over the network.

5 MITIGATION OF CVE-2016-0777

To demonstrate the effectiveness of ELFbac, we looked at CVE-2016-0777, the “roaming bug,” within OpenSSH 6.4p1. As stated above, the primary issue was an information leak within the roaming code that could result in cryptographic key disclosure. This is a mismorphism: access that was not possible in a programmer’s mind is possible in reality. Conceptually, our goal was simple: to isolate the network code from the crytographic key material. ELFbac is the intuitive means of expressing this isolation as a policy goal.

Retrofitting policy into existing software of any complexity can be painful. However, thanks to OpenSSH’s modular architecture, ELFbac policy and enforcement can be realized with minimal changes. First, a code review was required to identify the code

[3] The code for this linker can be found at https://github.com/sergeybratus/elfbac-arm/tree/master/tools/elfbac-ld

modules and functions related to cryptography and network communications. Once the requisite code and data are found, a policy can be crafted to define and isolate the desired relationships.

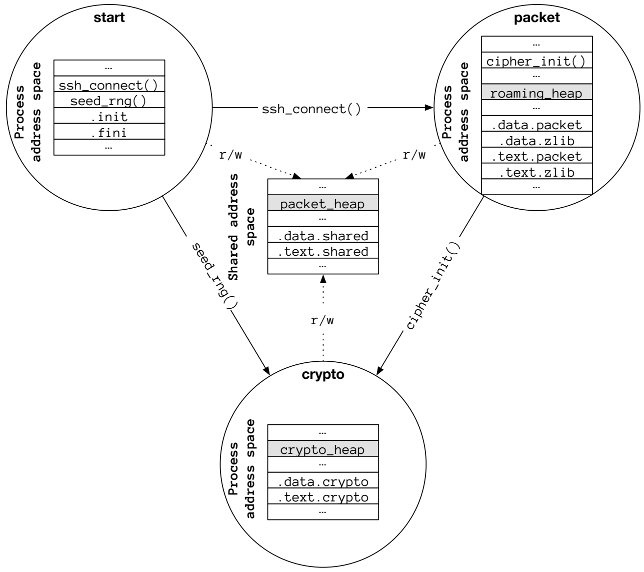

Figure 3: The ELFbac policy for CVE-2016-0777. Three states (start, packet, and crypto) represent the various phases of program execution. Each state is associated with a private and shared process memory layout. This dissection allows ELFbac to enforce separation between crypto-related memory and network traffic.

Figure 3 describes an ELFbac policy FSM compatible with the existing composition of OpenSSH that isolates cryptographic code and data from network communication. The policy does not attempt to enumerate all possible states within OpenSSH, rather we limit the scope to only those states of interest to the vulnerability at hand. The *start* state represents any setup and initialization that OpenSSH would normally perform, as well as the default state for execution. Code and data related to network communication are given a separate state, labelled *packet.* Finally, a *crypto* state collects the cryptographic code and data related to, for example, encryption and private keys.

Each state maintains a private view (or "shadow") of the process' memory, including private heaps. This is the primary method of isolation within ELFbac. In addition, a separate view of the process address space is created to allow any necessary data sharing. For example, a *packet_heap* is created that allows code within the *crypto* and *packet* states to communicate. There are many function call transitions that exist within OpenSSH. For brevity and conciseness, only a few function calls that trigger state changes are shown.

Figure 4 shows the policy in action, as we walk back through the information leak demonstrated prior. To the server, ELFbac's protections are transparent. To establish the remote connection, OpenSSH will transition between the *start* and *packet* states with *ssh_connect().* Part of this connection will require authentication, which means a transition into the *crypto* state. At this point, cryptographic keys have been written to memory within the crypto heap, inaccessible to any other states. Similarly, once the connection has been completed, the client-side roaming buffer will be stored within the roaming heap, again isolated from other states. Upon a disconnect, and subsequent resumption of session, the server may request as much of the buffer as desired. While it may be poor practice to ignore consistency checks between the client-side data-written versus the amount of buffer requested, the result is unaffected. Cryptographic keys no longer share the same memory space as the roaming buffer, and the information leak is mitigated.

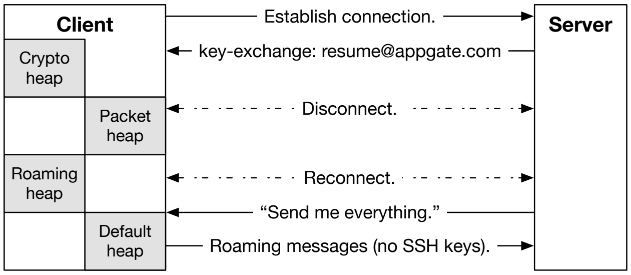

Figure 4: The information leak in OpenSSH with ELFbac's memory isolation policy in place.

Policy Definition

The policy is defined in JSON, using a format familiar to anyone with knowledge of linker scripts. Each state is defined, with the associated sections created or shared appropriately. Listing 1 is a condensed view of the policy state and transition definitions.

Due to this breakdown, any cryptographic keys stored will be on the crypto-state heap. In the event that a rogue server is able to perform the exploits necessary for the roaming vulnerability to trigger, any attempts to access the crypto-heap from packet-code will be caught by ELFbac, resulting in a segmentation fault.

To utilize the newly created, custom state-heaps, OpenSSH's buffer initialization routine must be made "heap-selective." This requires the addition of a custom memory manager; however, the only functionality that was used was that required to wrap the existing *buffer_init()* function call found in OpenSHH's buffer.c file. The wrapping allows buffers to be initialized on the custom created heaps. Listing 2 shows the necessary code changes required to wrap *buffer_init().* Now, heap selection is as simple as replacing a single function call. Only two files, authfile.c and packet.c, required this selectivity.

With a policy defined, all that remains is to annotate the necessary C-files within OpenSSH to utilize the defined state heaps. These annotations take the form of a common C-compiler pragma, *__attribute__((section(...)),* as seen below:

```
__attribute__((section(".bss.shared")))
int debug_flag = 0;
```

In total, 27 annotations in 4 files were all that was necessary to achieve the critical isolation. Our complete EFLbac-based patch can be found online. [1]

Listing 1: The JSON definitions for the ELFbac FSM policy state and call transitions.

```
 "states": [
  {
   "name": "packet",
   "stack": "stack",
   "sections": [
    { "name": "packet_heap", "description": "*(.
        data.packet_heap)", "flags": "rw" },
    { "name": "roaming_heap", "description": "*(.
     data.roaming_heap)", "flags": "rw" }
    ]
  }, {
   "name": "crypto",
   "stack": "stack",
   "sections": [
    ...
    { "name": "crypto_heap", "description": "*(.
        data.crypto_heap)", "flags": "rw" },
    { "name": ".data.packet_heap", "create":
        false, "flags": "rw" },
    ]}, ... ]

 "call_transitions": [
   {
     "from": "start",
       "to": "crypto",
       "address": "seed_rng",
       "param_size": 0,
       "return_size": 0
   }, {
     "from": "start",
     "to": "packet",
     "address": "ssh_connect",
     "param_size": 0,
     "return_size": 0
   }, {
     "from": "packet",
     "to": "crypto",
     "address": "cipher_init",
     "param_size": 0,
     "return_size": 0
   }, ... ]
```

Listing 2: The wrapping of buffer_init() in buffer.c by the ELFbac memory manager.

```
 void
-buffer_init(Buffer *buffer)
+buffer_arena_init(Buffer *buffer, enum arenas
     arena_idx) {
  const u_int len = 4096;

  buffer->alloc = 0;
- buffer->buf = xmalloc(len);
+ buffer->buf = xmemmgr_alloc(len, arena_idx);
  buffer->alloc = len;
  buffer->offset = 0;
  buffer->end = 0;
+ buffer->arena = arena_idx;}
+
+void
+buffer_init(Buffer *buffer){
+ buffer_arena_init(buffer, SHARED);
 }
```

6 ELFBAC POLICY CREATION

In general, the authors believe that ELFbac policy creation should take place during development. The programmer is best suited to create policy based on the domain knowledge and mental model already guiding the rest of development. Identifying code paths or data segments that must needs be isolated is already a priority. ELFbac policy captures the programmer's intent to isolate security critical sections within a program, and enforces the boundaries presumed by the common scoping model. Were ELFbac policy to have been utilized as part of the OpenSSH development process this roaming vulnerability would never have seen the light of day. Using policy to reclaim and enforce programmer intent for intraprocess interactions mitigates an entire class of bugs that depend on manipulating the broad interactions of code and data within a process.

As shown above, it is possible with minimal effort to retrofit an ELFbac policy into existing software. The process for doing this is largely one of intuition. It is tempting to begin with policy definition; however, in practice, it is likely infeasible to isolate every third-party library, function, and shared memory array. Familiarity with a codebase is essential to understanding potential areas of vulnerability. Not every codepath will be vulnerable, and not every piece of data is critical to an application's security. So, a first step to ELFbac policy creation is to identify *what* needs to be isolated.

The following step is to decide *how* isolation can be achieved. There may be no single answer at this stage. It is important to identify the default behavior of an application, e.g., which external libraries are being loaded and where they are being used. Common areas of concern may be input processing or memory buffers. As with OpenSSH's roaming bug, and so many other famous vulnerabilities, mistakes in input processing and buffer control can spiral into a multitude of unintended computations. Equipped with the knowledge of what needs to be isolated, a starting place may be a simple *grep* to identify relationships between an area of interest and functions, files, or libraries. Once relationships are identified, they must be codified into policy.

ELFbac only permits explicit relationships. As a result, policy creation can be an iterative process. Beginning with a simple set of states and relations, and refining acceptable transitions based on varied inputs and codepaths. Unfortunately, this process is not currently automated.

7 CONCLUSIONS

Mismorphisms between a programmer's mental model of software and actual program execution lead to vulnerabilities. Unintended computation due to such vulnerabilities results in exploitation. Explicitly codifying programmer intent into policy allows enforcement mechanisms to prevent exploits. The authors previously introduced ELFbac as a technique for specifying intent via kernel enforceable policies. In this work, we demonstrate the power and flexibility of ELFbac in mitigating a real-world software vulnerability, namely, the roaming bug found in OpenSSH.

For this work, we have produced a software patch to OpenSSH version 6.4p1 [1]. This patch is minimal in lines of code and allows the desired and necessary memory isolation between network and cryptographic code sections. In addition, we have written an ELFbac policy that defines the desired intraprocess isolation that we believe an OpenSSH programmer would have originally intended; that is, a separation between code that handles network connections and code that handles private authentication keys. The linker, loader, and kernel mechanisms of ELFbac work together to enforce this ideal scoping. Were these principles to have been utilized at design time, this unintended, experimental roaming feature could never have been weaponized and may have been identified sooner.

ELFbac shows tremendous promise as a mitigation tool. Many methods exist to identify and enforce privilege separation. However, ELFbac allows complex policy to be written in a simple to understand, commonly used syntax at the ABI level, with minimal changes to an underlying codebase. At design-time, policies are easily built and incorporated from a programmer's familiar understanding of scope. In addition, enforcement takes place within the kernel, utilizing the existing framework of virtual memory management.

Future Work

Work remains to further ease the burden of intuition in policy creation. Additional tooling with techniques from static analysis, such as data and control flow analysis, may be helpful in understanding the relationships inherent in extant software. While retrofitting software with ELFbac policy is not ideal, we have shown it feasible with minimal mental overhead.

As with any security primitive, there is likely to be a trade-off between performance and security. Analyzing the performance impact of ELFbac is situationally dependent on the software that is being protected, the hardware on which it is run, and the granularity at which the desired policy is implemented. Naturally, repeatedly forcing many heavy context switches will result in a significant performance penalty. In previous case studies, we have seen the performance degradation be as low as 3% with libpng on an AMD system, and as high as 30% with Nginx on an Intel system [6]. With software like OpenSSH, the performance bottleneck is bounded by the user interaction; otherwise, the performance hit should be negligible. Clearly, however, there is significant room for performance optimizations to ELFbac mechanisms.

Modern software development is a mix and mash of black-box pieces arranged in often precarious ways to achieve some ad hoc functionality. ELFbac's mitigations are as good as the modularity of a program. In fact, ELFbac is a simple mechanism of enforcing the intents expressed by a program's modularity. Library writers may find ELFbac an ideal method of isolating the internals of their code from its consumers. Similarly, application programmers can utilize ELFbac to more precisely specify the interactions of their code with that of utilized libraries. In this mutually-distrusting view of software development, the developers of each black-box component may define their own policies. Managing these various policies and their interactions is a challenging, but interesting task that we leave for future work.

Much of security policy has been predicated on the assumption of capabilities provided by hardware, namely memory isolation enforced by the processor and memory management unit (MMU). Page tables have typically been regarded as only a bookkeeping optimization. The contents of the page tables themselves were not considered policy objects, but rather artifacts in the mechanism of policy enforcement. Security policies may dictate access controls within a page table, but not the contents of the page table itself. ELFbac changes this paradigm by considering page tables as first-class objects of security policy, and using the existing page table mechanisms to isolate intraprocess memory.

Recent attacks exploiting previously unconsidered, latent microarchitectural effects have shaken the foundations of the aforementioned assumptions. The Meltdown [13] attack relies on out-of-order execution and kernel memory mapping. While ELFbac could be ported to the Linux kernel, that is not our current research direction. Luckily, Kernel Address Isolation to have Side-channels Efficiently Removed (KAISER) [11] does prove to be an effective mitigation. Unfortunately, Spectre [12] is a different story, not mitigated by KAISER. Spectre relies on speculative execution and specifically assumes "that speculatively executed instructions can read from memory that the victim process could access normally, e.g., without triggering a page fault or exception." We theorize that ELFbac may be able to partially mitigate variant 1 of Spectre by isolating those sensitive code and memory sections within a process. As seen in this work, isolating process memory and triggering exceptions is part and parcel of ELFbac's mitigating protections. We are pursuing this lead, but do not have conclusive results to share at this time.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the Department of Energy under Award Number DE-OE0000780. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

[1] Elfbac patch for OpenSSH 6.4p1. https://github.com/mjkoo/openssh-elfbac/compare/original...master.
[2] CVE-2002-0640. Available from MITRE, June 2002. Online at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0640.
[3] CVE-2012-2334. Available from MITRE, May 2012. Online at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2334.
[4] CVE-2016-0777. Available from MITRE, January 2016. Online at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0777.
[5] J. Bangert, S. Bratus, R. Shapiro, M. E. Locasto, J. Reeves, S. W. Smith, and A. Shubina. ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection. Technical Report TR2013-727, Dartmouth College, Computer Science, Hanover, NH, May 2013.
[6] J. Bangert, S. Bratus, R. Shapiro, J. Reeves, S. W. Smith, A. Shubina, M. Koo, and M. E. Locasto. Sections are types, linking is policy: Using the loader format for expressing programmer intent. BlackHat USA, 2016.
[7] D. J. Bernstein. Disabling the network. https://cr.yp.to/unix/disablenetwork.html.
[8] D. J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages

1–10. ACM, 2007.

[9] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. *;login:*, 36(6), December 2011.

[10] M. E. Carson. Sendmail without the superuser. In *Proceedings of the 4th conference on UNIX security symposium-Volume 4*, page 5. USENIX Association, 1993.

[11] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[12] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[14] NIST NVD. CVE-2016-0777. Available from NIST NVD, January 2016. Online at: https://nvd.nist.gov/vuln/detail/CVE-2016-0777.

[15] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[16] Qualys. Roaming through the OpenSSH client: CVE-2016-0777 and CVE-2016-0778. https://www.qualys.com/2016/01/14/cve-2016-0777-cve-2016-0778/openssh-cve-2016-0777-cve-2016-0778.txt, January 2016.

[17] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[18] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[19] S. W. Smith, R. Koppel, J. Blythe, and V. Kothari. Mismorphism: a Semiotic Model of Computer Security Circumvention (Extended Version). Technical Report TR2015-768, Dartmouth College, Computer Science, Hanover, NH, March 2015.

[20] I. Thomson. Evil OpenSSH servers can steal your private login keys to other systems – patch now. The Register, January 2016.