ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection

Julian Bangert, Sergey Bratus, Rebecca Shapiro,
Michael E. Locasto*, Jason Reeves, Sean W. Smith, Anna Shubina

Computer Science Technical Report TR2013-727
Dartmouth College

June 14, 2013

Abstract

Adversaries get software to do bad things by rewriting memory and changing control flow. Current approaches to protecting against these attacks leave many exposures; for example, OS-level filesystem protection and OS/architecture support of the userspace/kernelspace distinction fail to protect corrupted userspace code from changing userspace data. In this paper we present a new approach: using the ELF/ABI sections already produced by the standard binary toolchain to define, specify, and enforce fine-grained policy within an application's address space. We experimentally show that enforcement of such policies would stop a large body of current attacks and discuss ways we could extend existing architecture to more efficiently provide such enforcement. Our approach is designed to work with existing ELF executables and the GNU build chain, but it can be extended into the compiler toolchains to support code annotations that take advantage of ELFbac enforcement—while maintaining full compatibility with the existing ELF ABI.

1 Introduction

This paper presents the design and implementation of *ELFbac* (*ELF behavior access control*), a mechanism that separates different components of a program's memory space at runtime and polices their interactions to enforce the intended pattern of these interactions at the granularity compatible with the program's ABI format units such as ELF sections.

Modern programs are primarily built by composing libraries, modules, classes, and objects into some cohesive whole. Each component, composed of code and/or data, has some explicit high-level purpose or *intent*. The intended use of a component defines its *expected behavior*. Most cases of untrustworthy behavior result from a program's executable violating the programmers' intent.

For example, the intended usage of the `libpng` library is to convert compressed images to bitmaps; the intent of a table of function pointers is to be set up at the beginning of a process's execution and called throughout. The function pointers should *not* be overwritten by `libpng`—but that's what an attack exploiting a vulnerability in `libpng` may do.

*University of Calgary

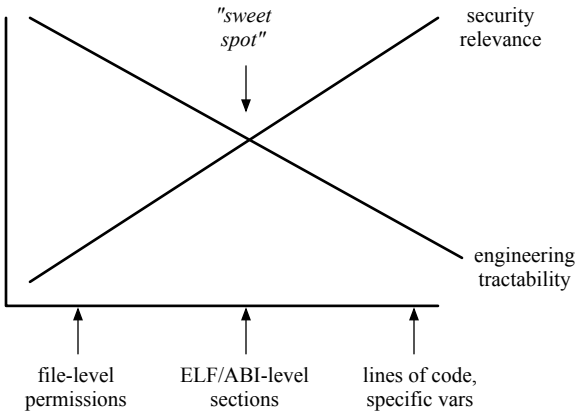

Figure 1: Looking at ABI-level code/data granularity hits the "sweet spot"

The standard process of programming takes code through the compiler-linker toolchain into an executable format, which is then interpreted by the OS loader and maintained by the dynamic linker and OS throughout the program's execution. In this paper we focus on *ELF*, a standard *executable and linker format* in the Unix OS family, but our ideas can be applied to other types of executable formats and operating systems.

Our core idea is that data structures in this executable and library format provide a natural, flexible, and effective venue to express and label code and data components that act as our policy subjects and objects. We use this idea to develop a prototype of an ABI (application binary interface)-compatible loader and hardware-assisted runtime mechanism that allows the developer to specify intent-level access semantics for ABI-described code and data units—and then prototype a way for the kernel to enforce these semantics.

Existing software security mechanisms tend to target a single granularity for objects and enforcement. Traditional Unix security policies operate on a whole file/process; SELinux policies operate on system calls. We argue that usable and more expressive security mechanisms must be able to adapt to the complexity of the software engineering effort and must be able to target the different intents of its components. As Figure 1 shows, this is the "sweet spot" for policy enforcement as it allows for detailed, yet not burdening communication of a component's intent from a programmer to the enforcement mechanism throughout the binary toolchain.[8]

This Paper Section 2 discusses why previous approaches insufficiently capture the programmers' essential intent for interactions between semantically different units of code and data. Section 4 reviews ELF. Section 3 explains our approach to policy. Section 5 presents our design. Section 6 discusses our prototype. Section 7 sketches out ways to evaluate its real-world effectiveness. Section 8 discusses future work, and Section 9 concludes.

2 Prior Approaches to Access Control

2.1 Motivating Example

Large numbers of software vulnerabilities arise because code and data that live within the same process address space can interact with each other in ways not intended by the developer. We consider a simple example of unintended interactions to highlight the shortcomings of current approaches to access control.

```
static char *encryption_key = "super secret";

void input(){
  int *data = read_data();
  process(data);
}

void process(int *data){
  data[data[0]]= data[1];
  /* We have a trivially exploitable
   * write-1-byte primitive here.
   * In the real world, this would be hidden
   * or constructed through smaller primitives
   */

  void *encrypted_data = encrypt(data);
  send(encrypted_data);
}

void *encrypt(int *data){ ... }
void send(void *data){
  printf("%s", data);
}
```

Figure 2: Simple program demonstrating the problem of too-coarse protection granularity: the vulnerability in `process()` compromises the secret key used by `encrypt()`, even if `encrypt()` is a separate and well-vetted library.

Figure 2 shows some C source code for a file server that reads in requests for files, fetches the files from disk, encrypts the files, and sends the encrypted files over the network.

Unfortunately, `process()` has a buffer overflow bug. Due to this bug, an attacker with control over the input to the file server can write to arbitrary memory locations. The attacker can use this ability to change the flow of control in the server. If the server uses defense mechanisms such as the NX-bit preventing code injection, the attacker can use techniques such as return-oriented-programming [21]; if the server uses the defense address space layout randomization (ASLR), the attacker can use techniques such as Müller's [20].

Once the attacker alters the control flow via this bug in `process()`, the attacker can proceed to the "crown jewels" in `encrypt()`, such as disabling encryption or extracting the encryption key.

It is not unreasonable to assume that `encrypt()` was provided by a well-vetted third-party library. Finding a vulnerability in such libraries that leaks the cryptographic key material is often quite difficult, and if such a bug were found, it would garner enough attention that the vulnerability would be fixed quickly (e.g., as evidenced with the response to the OpenSSH key bug [27]). Nevertheless, because all of these code and data elements live within the same address space, the key material can also be leaked through exploiting the `process()` function, which probably would not have been (and should not have needed to be) analyzed by the more security-aware engineers who implemented `encrypt()`.

In Section 7.2, we will demonstrate security vulnerabilities that result from treating all code and data in a process's address space equally and show how ELFbac prevents these vulnerabilities even with simple policies.

2.2 Traditional OS Memory Approaches

In the standard approach to runtime memory protection, the operating system uses the memory management unit (MMU) to ensure that processes cannot access each other's code and data by providing each process with its own virtual memory address space. This mechanism also keeps userspace code from directly accessing kernelspace code and data. MMU protections can also ensure that page-wide memory regions intended to be read-only cannot be written, and (more recently) that writable memory cannot be interpreted and executed as instructions. These memory protections provide a surprising amount of safety for their simplicity; however, by themselves, they cannot prevent userspace code from illegitimately writing to writable memory within the same process (as happens in the example from Section 2.1 above).

2.3 Traditional Unix-family File Permissions

Traditionally, Unix system administrators set access control policies at the file granularity: files are access policy objects, user IDs are subjects (expressed as effective and group IDs as domains), and (in the basic view) {read,write,execute} are actions. The subjects and objects are labeled with integers representing user IDs. The operating system itself propagates object labels using a set of strictly-set but easy-to-understand rules: in the absence of the `setuid`/`setgid` bits, a new file's label is determined by the label (effective user ID) of the process that created the file. Unix access control rules are typically based on the user ID of the process, user and group ownership, and permissions of the file. In the most typical case, the user-owned process accesses the files owned by the user.

Under this initial course-grained model, it up to the system administrator to label users and files in a way that minimizes the damage a single subject can inflict on the rest of the system. Unfortunately, this traditional approach does not reach within a process's fine-grained data structures (as objects) or fine-grained code structure (as subjects)—thus still permitting the attack of Section 2.1.

However, as we step through the evolution of Unix-family security, we see a trend toward intra-process controls.

2.4 Traditional Unix-family Process Permissions

Files aren't the only objects that require access protections or controls. Most other decisions that need to be made, such as whether a process itself can `kill` another process, are made by some mandatory access control logic baked into the kernel that makes a decision based on the processes' labels. This system of labeling acts as a low-overhead coarsely-grained *sandbox*, confining the reach of a subject's actions.

Again, these types of protections are unable to protect a process from itself, as happens in Section 2.1.

2.5 Newer Unix-Family Approaches

We posit that there are issues with such coarsely-grained protections. For example, whether a process has access to privileged operations (such as opening privileged ports) is based solely on the process's label being that of the administrator (user ID 0). If a process needs to bind to a privileged port, it needs to run as user ID 0, which gives privileges to perform *any* privileged task including the ability to read and write to *any* file. Thanks to the coarse granularity, the administrator had to decide either to fully trust that piece of software or to not use the software at all—no middle ground was possible.

`setuid` Thus Unix moved in the direction of allowing privileges to be dynamically and systematically revoked from a subject. Developers started designing software using `setuid` for dynamic privilege revocation; after it completed its privileged operations, a process could call `setuid` to change its user ID to that of a non-administrative user—thus limiting damage should it be compromised after that point.

The `setuid` method of privilege management is a sort of intra-process access control in that the capabilities of a given piece of code executing as part of the process could change during the process' lifetime.

The number of policy controls available to a UNIX administrator eventually grew, allowing for more flexibility: administrators gained the ability to set filesystem usage quotas by file label—controlling things such as scheduling priority, maximum number of files that can be open at once by subject label, maximum number of processes that a process with a particular label can spawn, etc.

SELinux With SELinux [16], administrators were given the ability to label and control access to objects at an even finer granularity. Not only could files be labeled, but so could file descriptors, filesystems, IPC message queues, and other objects. Furthermore, different types of files can be treated differently even if they harbor identical labels: a socket and a directory can share the same label but an administrator could write rules to allow a given subject to write to the directory but not to the socket. Actions that could be allowed and disallowed could flexibly depend on the class of object; for example, if the object is a socket, actions including `bind`, `listen`, `read`, and `write` can be allowed. All of this was part of a larger vision—to allow a given subject to perform only the system-related operations it needed for operation and nothing more, in a label-based low-overhead configurable sandbox.

Administrators were also given more control on the propagation of labels between subjects as well as between subjects and objects. A subject's label can only be changed during a call to `exec()`, but whether this label transition happens depends on the labels of the subjects and objects involved. By default, newly created objects (files, etc) inherit labels from their creator, but this behavior can be flexibly overridden in policy.

Nevertheless, even though SELinux provides administrators with a fine-grained way of controlling how a process can interact with other objects via system calls, it is not able to protect a process' data from the process itself—the problem of Section 2.1 remains.

PRM Solaris' *process rights management* (see Chapter 5 in [17]) offers a more fully fleshed-out approach to fine-grained privilege management. Solaris PRM allows administrators to add and remove a large set privileges from any user and executable. Privileges that can be managed include: calling `exec()`, sending signals to or tracing other processes, accessing the network, changing ownership of files, and binding to privileged ports. Solaris also provides an API so developers can build privilege-aware applications, allowing for processes to dynamically enable and disable their own privileges and stepping closer to our goal of not treating all code and data mapped into a process equally.

However, the privileges that differ between sections of code are only privileges directly associated with system calls—`bind`, `read`, `signal`, etc. In contrast, our ELFbac is able to restrict privileges beyond those guarded by system calls.

2.6 Other Prior Work

Access control in the Unix-family has evolved to protect the system from misbehaving applications by limiting each application's privileges. But even with these security mechanisms in place, exploits were still able to cause damage.

In the early 2000s, efforts were also put into preventing application compromise by addressing the mech-

anisms leveraged by exploits. These efforts included *DEP (data execution protection)* and *ASLR (address space layout randomization)*, both pioneered by the PaX project [26]. By preventing writable memory from being executed, DEP prevents a process from directly executing code masquerading as exploit-injected data. By randomizing where libraries are loaded in a process' address space, ASLR makes it harder for an exploit to reliably depend on libraries in use to execute its payload.

Even with DEP and ASLR in place, any piece of code in a process can access any data in the process' address space. Various proposals for granular memory protection (e.g., early [4, 12] and more recently MIT's *Mondrian* [29, 30]) leave the fundamental plight of a Unix or Windows engineer who has to use a large standard library for a small and data-specific task un-addressed in practice: *all* of the imported library code implicitly has access to *all* of the program's data (including sensitive bits such as keys or certificates). To the best of our knowledge, we are the first to expose this problem as a matter of severely lacking the right granularity to express and enforce intent.

UNIX mechanisms such as BSD jails, SELinux, and AppArmor provide system administrators with some means of specifying "this daemon is only expected to access these files and directories," but offer no way for programmers to specify the same thing about a library—because when it comes to security, all code in a process' address space is treated equally.

Some developers have taken the route of building sandboxes directly into their software using existing OS-level protection mechanisms to achieve a level of separation between code units running as separate processes. The Chromium browser was designed from the ground up to employ this type of sandboxing [5]. ELFbac allows for code and data segments to be isolated from each other without needing to spawn extra processes or re-engineer code.

In summary, our proposed system describes runtime measurement and enforcement of the programmers' intent at different levels of granularity, which roughly correspond to the layers at which runtime computation is engineered. The units used at different layers mesh (are "stacked") across layers and are intelligible to the build toolchain and the runtime ABI. To the best of our knowledge, this *cross-layer* approach has not been explored in literature. For example, CFI [1] concentrates on one level of modeling software and ignores the ABI; traditional semantics methods concentrate on a more granular level of programming language abstractions [15], and so on.

Trusted Computing. Last but not least, several projects combined memory virtualization with Trusted Computing Group's (TCG) architecture, segmenting memory into measured regions according and protecting these regions by means of virtual machines or hypervisors. In a sense, such combination is a necessity, since the TCG architecture as such does not deal with the essential TOCTOU threat (as we pointed out in [6]), and must rely on a runtime memory protection mechanism to maintain its integrity. However, we believe that aligning memory measurement and protection design with the existing ABI is crucial for deployment prospects of a system; hence we focus our attention on the ABI and OS memory artifacts and memory trapping rather than on the additional opportunities provided by the TCG architecture (but see [8, 9] for our discussion of possible ABI-aware TCG-backed policies).

Flicker [19] uses AMD's Secure Virtual Machine extensions (SVM) to allow an application to verify and run code in an isolated environment, also making use of the Trusted Platform Module (TPM) sealed storage to save state between different executions; Flicker leverages SVM and TPM to remove implicit trust in the OS. A Flickr kernel module offers an interface for applications to make use of the SVM extensions.

Fides [25] builds on the design pioneered by Flicker to develop the concept of *protected modules*, with protection enforced by a dedicated hypervisor. Fides uses a program-counter based access control mechanism that guarantees restricted entry points, protection of module data, as well as mutual authentication by modules and secure communication between them. Fides also leverages the hardware to exclude most of the OS from its TCB, which includes the hardware, the Fides architecture itself, and the module and its communicating peer modules. A module runs in the same address space as its application; the application's

code running outside of a module gets only limited access to the module's memory. Fides' concept of using the process counter to characterize the trust level of currently executing code maps well to our concept of code sections corresponding to process execution states that are governed by different (ABI-aligned) memory access rules; however, we decouple these states from code sections, so that the same code section may be executed in different states, depending on the previous history of state transitions in a process.

3 Our Approach

In this prior work, we see a trend of restricting of what an application can do *as a whole*. However, there is a clear benefit for, within an application, not treating *every segment of code* (including all the libraries) as equal. Just because `fork()`, `exec()`, and `memcpy()` are mapped into a process's address space doesn't mean security-critical code that sanitizes user input should be able to invoke these functions at any time.

We believe the solution is to move toward compartmentalizing segments of code and data inside a process's address space. Our approach is to use the code and data granularity *already present* in the loader format to specify intra-process security policy, and then to augment the existing software and hardware architectures to enforce it. We believe that this approach will lend itself to developing policies supported intent-level semantics.

3.1 Intent-level Semantics

Our goal is to help humans and systems better reason and communicate about trustworthy behavior. A basic characterization of a trustworthy system is "the system and all of its components behave as *intended* and *expected*." Thus, to build a trustworthy system, it is crucial to be able to express and communicate both *intentions* and *expectations* as well as verify their congruence (or at least compatibility). One of the simplest and yet most important elements of this communication is specifying *intent* of a particular software component—that is, a high-level description of *what it is meant for*.

An application's developers typically have the best understanding of what the intended behaviors of the application's code units are, even within large multi-developer applications. An application's developers are also best suited for picking out what variables/symbols/data/control flows are most sensitive and important with regards to security. For example, the developers tend to have the best idea of which sections of code should be off-limits until a user has authenticated, and which sections of data should be off-limits to external helper libraries[1] throughout the program's runtime. As a result, we have designed ELFbac with the purpose of allowing an application's developers to express an application's intended behaviors in an ELFbac policy in a concise yet usable manner.

3.2 Policy Labeling Granularity

The labeling of code and data units for programmer intent/expectation must exist and be enforced on the same level where the primary act of software engineering—*composition*—takes place. Security labels must match the units in which software is written, imported from libraries, and, ultimately, loaded and serviced at runtime. ELFbac flexibly supports labeling at different granularities, from individual symbols to whole libraries, so a policy can be expressed at the granularity that makes the most sense to the developer.

These days, it is practically impossible to write a meaningful program without using code from scores

[1] Attackers leveraged vulnerabilities in popular image and media libraries to get through them at the application's "crown jewels" data, even though those libraries were obviously not intended to access such data, only to render generic presentation material.

of different external libraries, likely written by third-party developers. Even if the original libraries were vetted in some way, they will likely be updated or replaced as the underlying operating system and system configuration changes. Furthermore, even first-party code is likely somehow structured into different modules each with their own intended uses. For example, consider the modern Firefox browser. While it is running in Linux, over 100 libraries are mapped into its address space and can be accessed from any code executing in the context of the Firefox process. Examples of libraries mapped into the Firefox address space include `libFLAC.so` (encoding/decoding certain types of audio files), `libdrm.so` (direct access to video hardware), `libresolv.so` (handles DNS lookups), `libssl3.so` (SSL and other crypto), and `libc` (standard C library functions). Each library has its own purpose and is used by Firefox with a certain intent in mind. However, a bug in one library (for example `libdrm.so`) can triggered by a second library (for example, `libresolv.so`)—even if, from the developer's point of view, the second library has no business accessing the first library. Since Firefox uses each of these libraries with a certain intent in mind, it is natural to draw trust boundaries around each library and within main the executable itself, treating each segment of code and data within a boundary differently from a security perspective.

There are also other potential natural boundaries at various granularities we can consider—including functions and object files, as each are created with a certain intent in mind.

However, all these things—functions, object files, libraries—have two key things in common:

- Each is based on a program's *intent-level* semantics. [2]
- The compiler-linker toolchain has knowledge of each during the build process.

In ELFbac, we use the latter to capture the former.

In this paper we focus on behaviors and relationships that are expressible in the form of behavior access control at the granularity of ELF/ABI units, such as libraries and memory regions, that have a simply-defined programmer intent. Examples include "this library function is only meant to work on this kind of data, type of object, or region of memory", "this library function is only intended to be executed during initialization phase of a program or during error-handling," or "these data are only meant to be accessed in the initialization phase of a program or during error-handling."

3.3 Policy Creation

We need to be able to create policies that support intent-level semantics on the appropriate code and data granularity. To achieve this goal, we rely on a series of insights:

- Code and data are already semantically divided at development.
- These code/data divisions are conveniently carried over to the binary level by the loader format specification.
- Even if the developer does not explicitly specify a policy, we can often infer an implicit policy from the code/data divisions already in the executable's ELF section headers and symbol tables.

Developers tend to structure their applications into different modules and formulate "interface contracts" between the modules that specify the intent and behavior of each unit of code. Such rules/contracts can be

[2] In standard theory usage, program semantics are described in terms of predicates, invariants, and various formal logic statements. Such formal tools, however, are typically outside the reach of an industry programmer, at either developer or architect levels. We do not intend to use the term in this meaning and instead focus on high-level properties and statements that match common programmer intuitions.

explicitly encoded in object-oriented languages such as C++ and Java, where classes, fields, and methods are marked as public, private, or protected. It is up to the compiler to enforce these rules—unfortunately, *this knowledge is discarded and thus not enforced at runtime.*

Since software is designed in terms of libraries, modules and individual symbols, our policy mechanism targets these as granularity levels. We focus on the binary toolchain because it operates at precisely these granularities. Libraries and modules correspond to different ELF files, and most compilers allow the programmer to arbitrarily group symbols into sections through vendor-specific extensions at granularities that make the most sense to the developer. For example, in Figure 2, the programmer can write a very fine-grained policy for security critical components such as the `encrypt()` function, yet treat the entire `process()` module (which in a real world application would consist of many functions) as a single entity.

3.4 Unit Relationships

It's also important to consider the *unit relationships* between code and data. In a well-designed program, exclusive access relationships alone can serve as a good proxy for the unit's intent—and therefore as a basis for succinct expression of an enforceable access policy. (As we show later in Section 4, the standard ELF section structure already expresses a wealth of such relationships.)

Due to the inherent complexity of modern applications, we treat code units as black boxes whose intended behavior can only be verified by their interaction with other code and data units. Non-trivial semantic relationships can be extracted from mere annotations of intended accesses, just as a grammatically correct sentence made of nouns of obscure or undefined meaning nevertheless conveys important information about the referents of these nouns. A classic example is *"the gostak distims the doshes:"*[3] even though we don't know what the *gostak* or *doshes* are, we know of their relationships that the *gostak* is that which *distims* the *doshes*, that *distimming* is what the *gostak* does to the *doshes*, and so on. For all intents and purposes, we can now define a policy to describe this as an intended behavior, so long as we have labels for *gostak*, *doshes*, and can recognize *distimming*.

A complex program contains many code units that will likely remain as opaque to a policy mechanism as the *gostak* is to English speakers, and whose intended behavior can only be described based on their relationships with other units. This opacity strongly correlates with the amount of context needed for a code unit to determine if the data it is about to work on is valid. For a parser (or an input recognizer in general), it is comparatively easy enough—or should be easy enough with the right messages format design—to check if a particular input bits or bytes are as the input language grammar specifies, based on locally available context (e.g., a few bytes previously seen in input or a few bytes of look-ahead).

Unfortunately, not all code units are so comparatively lucky. Consider a typical collection data structure such as a heap or a hash table of C structs or C++ objects that are chained (via pointers or offsets) internally and with other objects; and consider code units that follow the pointers and change the collection elements and their linkages. For such a structure collection to be valid ("as expected"), it does not suffice for just the bytes of each particular member to be valid—all of the pointer (or offset) linkages must also be valid. Should a link point to an invalid location, a method that uses that link to mutate its target object will likely mutate something else, lending itself to exploitation use (as demonstrated by a large amount of hacker research, e.g., [18, 2, 14] and the most recent [3]).

This poses a conundrum for the programmer writing such code units. To make sure that all the linkages are correct (for the semantics of data structure), one would potentially need to walk them all, which is unrealistic, inefficient, and would explode a simple task such as adding or deleting an element from a heap free list or a hash table's collision list into verifying a lot of things about the entire table first. In other words, the context needed to verify the data being worked on is too large and non-local. So this is not how

[3]The phrase was coined in 1903 by Andrew Ingraham, and popularized by C.K. Ogden and I.A. Richards in their book *The Meaning of Meaning.*

such code is written. This brings us to how it's actually done.

A typical accompanying expectation for these code units is that they are substantially isolated from (properly validated) inputs—otherwise maliciously crafted input easily leads to numerous heap meta-data and object store exploits that make exploit primitives out of innocent object maintenance routines. Therefore, when programmers write such code, they make believe that all the previous invocations of related code units have left the data units in valid linkage states, and *no other code units have touched it.* In other words, the programmer's expectation of input validity are not checked byte-for-byte, but rather are based on the isolation of the data and its orderly access only by specifically allowed code units working in concert with the one being written—that is, it is based on *exclusive access relationships of code and data units.*

Broadly speaking, programmers conceive both intent and trustworthiness of these code units in terms of data flows from and to these units. Although a simple input parser would know if the next byte is wrong because of the previous bytes, a data collection maintenance method such as a heap manager or hash table manager method cannot realistically visit and check all the bytes that matter for determination of validity. So the granularity level of programmer expectation is, "no other code has touched this heap segment".[4]

This expression also has an attractive relational duality[5]) in its expression of intent semantics: intent of code is defined by the data it is expected to access, and vice versa. To the best of our knowledge, this approach has long been neglected (at least since the demise of segment-based programming models and tagged architectures), with the recent exception of *SegSlice* [7].

Coming back to our *gostak* example, an intent-level policy would play the role of the grammar to express the ubiquitous programmer expectations described above—expectations ubiquitously ignored at runtime. Only the labeled *gostak* may *distim* the labeled *doshes*, and it's only the *doshes* that are being acted on by the *gostak.* This is all we know, and it is enough for maintaining a level of programmer intent. All of these entities are opaque, as well as their intended actions, but we trust the whole so long as no other actors and actions are in the picture. A heap metadata chunk is only to be operated on by heap manager, and a heap manager is what operates on heap metadata chunk. If it *distims* the *doshes*, it's the *gostak*, and the *gostak* is what *distims* the *doshes.* Should *drokes* or *duskats* attempt to act on the *doshes*, we know something gets untrustworthy about the *doshery.*[6]

We can draw an analogy to what's called *duck-typing*: if it read/writes duck data, it's a duck, and that's all we know. Then we describe the duck code to the runtime reference monitor in these very reduction terms: enforce such relations, keep non-duck-labeled code from touching duck-labeled data, let only the marked *gostak distim* the labeled *doshes.*

3.5 Memory Flow Barriers between Code Sections

One important aspect of enforcing the exclusive relationships between code and data section is *isolating code from data that it is not expected or intended to consume.* Indeed, programmers assume such isolation when they first apply sanity checks and transformations to input data, and then write subsequent code under the assumption that the data now has been "cleared" and can be operated on safely. The attacker's potential ability to slip in other, un-cleared data to code that expects cleared data likely breaks its explicit or implicit assumptions and leads to exploitation by maliciously crafted data payloads. Indeed, effective lack of such isolation underlies most modern "no-code" exploitation scenarios for binary executable targets.

[4]Note that hardened heaps such as Microsoft's Low Fragmentation Heap use canaries and XOR-ed check areas within the heap to create *local* validation contexts for heap manipulator code units that comprise the heap manager, as tell-tales of other code units' forbidden access; the security assumption there that such an unexpected access will trip the tell-tale and will invalidate the few bytes close (and check-able) to the pointers or offsets being mutated by the manager code unit. Thus these hardened heaps bring granularity of checkable local context back towards the parsers' end of the spectrum.

[5]"Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious." (F.P. Brooks, *The Mythical Man-Month* [10]

[6]These additional entities appear in the interactive fiction "The Gostak" by Carl Muckenhoupt; it is freely available online at several interactive fiction sites.

As an example, consider a parser that is intended to scan the input data and either "sanitize" or "normalize" it, or to reconstitute it and build trusted data structures for further processing. Once that parser is done, none of the raw data in its input buffers should be accessible by subsequent parts of the process if these are meant to consume only "sanitized" and reconstituted data. However, in multi-stage exploitation scenarios that rely on parser bugs for their initial stage(s), that raw input data contains payloads for further stages. These payloads are brought into play by triggering the initial parser bug; blocking the data flow from raw buffers to vulnerable code would break the execution of the exploit.

Such a "memory wall" was introduced by The UDEREF part of the GrSecurity's PaX[7] hardening patch for the Linux kernel. Empirically, this wall has dramatically reduced the incidence of privilege elevation attacks. The kernel programmers' clear intention is for the kernel code to access only such user memory contents that have been validated, e.g., passed the appropriate checks at the start of a system call; lower layers of the system call's implementation rely on such checking ensuring coherent and well-formed data structures. Consequently, exposing kernel code to data structures constructed by the attacker in userland led to exploits: kernel code executing at the highest privilege corrupted kernel memory while operating on the crafted userland data. UDEREF[8] prevented a large number of such unintended data flows by blocking the kernel from accessing most of userland memory pages; the result was a dramatic reduction in exploitability of the hardened kernel. Notably, the new SMAP feature of x86 processors opens a way to more efficient implementations of UDEREF.[9]

ELFbac applies the same principle to the ABI-granular, intention-level semantic division of a process' code and data units.

3.6 "Forgetful loaders" Discard Implicit Intent-level ELF Semantics

Loader formats such as ELF already have implicit intention-level access semantics at the granularity of their code, data, and metadata sections. In ELF, these semantic relationships are partially expressed in the section header table fields (e.g., in ELF's *Info* and *Link* section entry fields), and partially implied by standard section name pairings (such as *.init* and *.ctors*, *.fini* and *.dtors*, *.plt* and *.got*, and others).

Unfortunately, under the current UNIX implementation conventions, the OS loader forgets all this information once it creates the process' runtime virtual address space. We call this the **forgetful loader** problem. In particular, the OS loader is driven by the *segment* header table rather than by the section header table. Segments aggregate many semantically different sections based merely on their memory read/write/execute profiles, and ignore other semantic differences.

For example, in ELF, the `.text` section is often grouped with the `.rodata` section into the same *"R-X"* segment because both sections are expected to be non-writable (and an attempt to write them indicates untrustworthy behavior); however, this is just about the only thing they have in common, semantically. The one important exception to this is the two segments that serve dynamic linking: the short `.interp` segment that contains the path to the dynamic linker-loader, and the `.dynamic` segment that indexes all sections referenced by the dynamic linker in its runtime operation.

Thus the loader, as it is currently conceived of, is the weak link in the chain of semantic information on ELF units from the source code pragmas, though assembler's and linker's native support of custom target sections. This appears to be a decision influenced by the paucity of OS/kernel and MMU support for discriminating between different program semantics units at runtime.

However, technically there is nothing preventing the OS loader and kernel from being as aware of ELF sections as the components of the development toolchain or the dynamic linker: the section header table is

[7] `http://pax.grsecurity.net/`, `http://pax.grsecurity.net/docs/`

[8] For the discussion of UDEREF's implementation details and efficacy, see `http://grsecurity.net/~spender/uderef.txt`

[9] See "Supervisor Mode Access Prevention for PaX", `http://forums.grsecurity.net/viewtopic.php?f=7&t=3046`

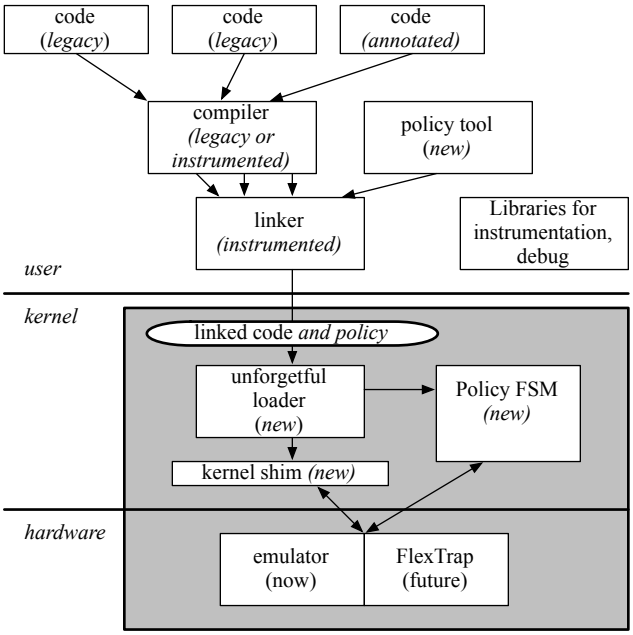

Figure 3: The ELFbac architecture

either typically mapped into the runtime address space, or can be trivially mapped so. Should the sections be relocated, this table could also easily be patched by means of the standard relocator implementation.

In fact, crucial section information is replicated in the dynamic symbol tables accessible to the dynamic linker through the `dynamic` segment, and is so made available in its context. This decision probably reflects the conceptual gap between the legacy, limited *OS loader/kernel*'s view of the runtime process semantic structure, and the more recent *dynamic linker/loader*'s view of this structure (which the latter manages).

However, these is no reason why we should stick to this disparity between an older and a newer design and keep the OS loader to a dumbed-down and forgetful view. Our vision is outlined in the next section.

3.7 Vision

Unforgetful Loader. Figure 3 sketches our resulting vision. Code chunks and libraries carry with them policies specified in terms of their loader formats. Developers can explicitly specify these policies to express their intentions—but for legacy code, we can also use the intentional policies already implicit in the loader format. Our **unforgetful loader** remembers this information, and a modified kernel enforces it.

4 The ELF Story

We quickly review loader formats in general and ELF in particular, for the reader unfamiliar with these topics.

4.1 Loader Formats

Binary executable formats have been designed to present a minimal description of the functionally different kinds of data (including compiled code) required for the OS to construct the process runtime image. These formats include special sections of metadata describing the extents and function of subsequent binary sections; these metadata sections serve as the instructions for the OS loader specifying the address ranges to copy the sections to, and the desired protections of the underlying memory pages.

By default, the loader performs a byte-for-byte copy of a continuous extent of bytes from the file to their appropriate places in RAM. When a simple byte-for-byte copy of the file's contiguous contexts does not suffice and an additional transformation of the loaded bytes is needed, other metadata in the file being loaded is needed to complete the loading process. For example, if some library code contains absolute addresses but the OS cannot load some library code at the starting address specified in its metadata, then the loader will need to transform the code by patching those addresses.

Binary formats evolved towards

- adding new types of functionally different data (so that the loader could be informed of the bytes' purpose and choose appropriate action), and
- including more detailed descriptions for internals of binary sections to enable more complex transformations.

Thus the binary format metadata evolved from mere loader copy instructions towards representing the semantics of sections by their intended runtime functionality, such as code for "initialization", "tear-down", "relocation", "dynamic linking", and the data to needed drive this code.

4.2 ELF

As the engineering of the Unix process' runtime became more sophisticated, ELF accommodated this trend by defining the necessary new metadata structures. In parallel, ELF added features to support more advanced debugging and exception-handling mechanisms. As a result of this co-evolution, a typical ELF executable binary contains over 30 sections, each which correspond to code and data units with different runtime semantics—expected behavior defined *in relation to other sections*.

This trend towards more detailed internal descriptions allowed ELF to unify the representation of binaries-to-be-loaded, and various intermediary stages of compilation and runtime. In particular, ELF is used to represent the compiler's object code files (".o"), shared libraries (a.k.a. shared objects, ".so"), and even core dump files.

ELF files are largely composed of ELF sections. An ELF section is simply a set of contiguous bytes in a ELF file and is described by an ELF section header, of which ELF maintains a table. Some ELF sections contain or describe the code and data of the program proper (`text` `rodata`, `data`). Others, such as `bss`, take no space in the file besides the entry in the section table. The `bss` section in particular describes the necessary allocation size for the program's uninitialized global variables. Other sections contain code and the necessary supporting data to be executed before the start of the main program or after its end (regular or forced) to set up or tear down the runtime. Examples of such sections are `ctors`, `dtors`, `init`, and `fini`, which control the calling of the constructor and destructor code for the objects in the runtime address space, including the necessary initialization of the runtime's own structures. (Here and later, we use the term *object* for a semantically distinct ELF code or data unit, rather than in pure OO or C++ ABI sense)..

The contemporary ELF design separates both the special purpose code and its data into separate ELF sections, named according to their semantics. This design decision creates a special and *exclusive* relationship

between these sections. A violation of this relationship at runtime is likely the sign that the process should no longer be trusted. If the implicit read, write, and execute relationships between these ELF sections were enforced, a number of exploitation techniques that use the contents of these sections contrary to their intended use and place in the process timeline would become infeasible. For a case in point, the data contained in `ctors` and `dtors` is only meant to be interpreted and acted upon by `init` and `fini`, respectively. A deviation from this access pattern may be a sign of runtime loss of trustworthiness. For example, Bello Rivas [23] demonstrates a method of exploiting a GCC-compiled program by making an entry in `dtors` that points to malicious code located outside the `fini` section.

Although this decomposition may have been originally motivated by abstracting away the algorithmically common part and the program-specific data part of the simple setup or teardown ("loop over a table of function pointers to constructors/destructors, calling each one in order"), the contents of `.init` need not be limited to a single simple loop, nor the contents of `.ctors` to a simple function pointer table, and the same for `.fini` and `.dtors`. As long as the contents of these sections agree, they will be true to their ELF-semantic roles with a more complex algorithm and more complicated data structures to drive them. This same argument can be applied more generally to any set or related code and data units.

PLT and GOT The `got` (global offset table) and `plt` (procedure linkage table) also show the special relationship between code and data ELF sections. The `plt` contains call stubs to external library functions to be linked at runtime by the dynamic linker, whereas the `got` contains the addresses through which these functions are indirectly called.

Initially indirectly calling a function through its `got` entry gives control to the dynamic linker which, with the help of this stub, is provided with an index of the function's dynamic linking symbol, including its name. After the dynamic linker resolves the symbol name and loads the appropriate libraries, it rewrites the `got` entry to point directly to the address where the function is loaded, so that further indirect calls through the `plt` result in calling that function with just a single indirection (through its `got` entry).

Although only `plt` is expected to read the `got` while dispatching through it, the dynamic linker is expected to write its entries. However, the rest of the code in the process' address space should not write `got` entries; in fact, any of its code unit doing so likely indicates a dynamic code hijacking [11, 28]. Notice also that any given `got` entry is meant to be written only once.

5 Design

Historically, the focus of ELF has been on the phases of a process' lifetime handled by various parts of the OS. However, the intent-related section specialization and typing need not stop at the OS view of the process. We believe section specialization can and *should* also express the developer's view of the intent of different program phases and their respective code and data segments, as well as the intent of any code borrowed from external libraries. We see this as the only possible foundation of a systems policy that is not blind to the programmer's knowledge of the expected and trustworthy program behavior.

The same ELF mechanisms used to describe loading-related data and code units are easily expressing the *program-centric* view of runtime, detailing the sequence and access expectations of the various parts of the program corresponding to phases of its process runtime.

As we discussed in Section 4.2, the ELF *section table* of a binary executable file contains a wealth of information about the *semantics* of its code and data units, such as the expected data access behavior of code units and the expected access profile of data by the code units. This behavioral information encoded in the ELF section table can be a basis of a policy enforcement mechanism enforcing these and similar kinds of expected behavior. In particular, a programmer interested in expressing behaviors such as "this code unit only accesses this data unit" or "this data unit is only accessed by this code unit at this particular phase

of the program's runtime" can easily put the relevant code and data units into specialized ELF sections. (The GNU GCC toolchain already supports creation of custom sections with the GCC extensions pragma `__section__` in the C/C++ source code.) With the minimal assistance of the loader, the memory management unit (MMU), and the operating system's MMU-generated fault handler these policies can then be enforced during runtime.

Section 5.1 presents our policy language. Section 5.2 discusses an ELFbac policy for our example from Section 2.1. Section 5.3 presents tools to assist a programmer in creating a policy.

5.1 ELFbac Policy

ELFbac policies are expressed as finite state machines (FSMs). Each state is a label representing a particular abstract phase of program execution that a given section of code in the program or library drives. The inputs to the automaton are memory accesses of the program. The policy defines a set of transitions between these states (such that there is at most one transition for each pair of state and memory address), and the allowed unit-level memory accesses for each state.

In particular, for each state in an ELFbac policy, there is a set of rules that specify what data and code can be used, and how these sections can be used in terms of read, write, and execute permissions. In this manner we are able to treat different sections of executing code within a single process differently with respect to security. These rules also need to specify what state to transition to after a given memory access, which often is the same state the process was in prior to the memory access. In a graph representation of the FSM, the transitions that do not change state correspond to loops, whereas those that do correspond to non-loop edges, so all information is encoded in the edges of the graph. In fact, the statements of our policy language are isomorphic to directed edges of the FSM.

```
policy_statement :=
     data_rule | call_rule
```

An ELFbac policy distinguishes between two kinds of rules: memory access rules and function call rules. Each policy rule is a transition in the finite state machine, so we call them "data transition" and "call transition" respectively. Each transition specifies a source and destination state and the interval of virtual addresses that trigger it. In any given state, there is exactly one rule that specifies how to handle a memory access to a given offset. If no rule is explicitly specified, the memory access is denied, which usually results in a segmentation violation and the (logged) termination of the program. If a rule does not specify a destination state, it is assumed that the destination state equals the source state.

```
data_rule :=
     from_state (-> to_state)
     {read,write,exec}
     from_symbol (to end_symbol)
```

Data transitions allow the program to read, write or execute the specified memory address. We expect most rules regarding data accesses to not cause state transitions—for example, "Code executed while in state *cryptography* is allowed to read memory labeled *key material*." However, rules that transition to a different state on a memory access attempt are also valid. If only a single symbol or address is specified, this corresponds to the implicit 'size' of the referenced object, as determined by the ELF symbol tables and headers. ("Code executing while state is *input* may trigger a transition to state *cryptography* when reading memory labeled *key material*"). Our current prototype has some restrictions on data rules that trigger state transitions, as detailed in Section 8.1.

```
call_rule :=
     source_state -> dest_state
     call symbol (return) (paraminfo)
```

A call transition does not allow memory accesses, yet its semantics make expressing transitions between states more convenient. It triggers on an instruction fetch from a single virtual address, usually corresponding to the entry point of a function. In addition to syntactic convenience features for parameter passing (which we elaborate in Section 8.2), call rules can also allow a one-time return transition back to the source state. Return transitions allow access to the instruction after the jump that went to the faulting instruction, corresponding to a function return.

If function returns were not treated differently, then a widely-used function (for example, in the C library) would have to be allowed to jump to the instruction following every call to it, which would give an attacker threatening flexibility in changing the control flow.

However, both the call and return transitions are merely convenience features and can be replaced by multiple states with data transitions when reasoning about the policy. For example, suppose we have states `X,Y,Z` with:

- `X exec x_func`
- `Y -> X call x_func`
- `Z->X call x_func`

We can then equivalently create two states `X_1,X_2` with:

- `X_1 exec x_func`
- `X_2 exec x_func`
- `Y -> X_1 exec x_func-x_func`
- `Y -> X_2 exec x_func-x_func`.

If this call rule had return transitions, those would correspond to

- `X_1 -> Y exec y_ret`
- `X_2 -> Z exec z_ret`

where `y_ret` and `z_ret` are the instructions following calls to `x_func`.

By removing these convenience features, the ELFbac policy can be reduced to a regular expression matching the memory accesses of a program. While it is very hard at best to make any useful statements about the machine code itself (being Turing-complete), we can make statements about the policy, as it's a weaker finite state machine. For example, it would be very convenient to be able to prove a statement such as "Data from the filesystem must be encrypted before being sent over the network." In general, proving such a statement about a program itself is undecidable by the Rice-Shapiro theorem [22]. However, assuming we can trust our cryptography and authentication code, we can put all code writing to the network in one state and then verify that all transitions leading to that state occur from the cryptography state. ELFbac then guarantees that all data will have been passed through the cryptography module before it is sent, reducing the amount of code that has to be formally verified.

5.2 Policy Example

Our example program in Figure 2 can be naturally divided into four phases: input, processing, encryption and output. In a reasonably well-designed program, these four phases would be in different modules, functions or classes, depending on the language in which the program was implemented. (We posit that, for programs in general, security-relevant progressing steps can be similarly separated into such stages.)

Figure 5 is such an ELFbac policy as the programmer creates it. Figure 4 shows this policy as a finite state machine. Each arrow label corresponds to one transition, which is triggered by access to a particular region in memory. Without ELFbac, only writable data and executable code are separated by the memory system, as shown by the shaded boxes.

The four states are only allowed to execute one of the code modules at once. Similarly, each has different access permissions with regard to the three data areas[10]. The input stage is allowed to access system calls[11] and write to the unencrypted heap area. The processing phase, which contains the critical security vulnerability, can only read or write to the unencrypted heap and neither affect the key material or cause sensitive data to be sent unencrypted.

5.3 Policy Creation Tool

Although we are actively working on more advanced ELFbac policy creation tools, we have a set of preliminary scripts that isolate a binary and the libraries with which it is dynamically linked. The policy this tool creates usually has two states, one for the dynamically linked libraries and one for the main binaries.[12] This tool analyzes ELF symbol tables to determine which imported library functions get called and creates a call rule for each of those. So far, the programmer manually needs to allow callbacks and other instances where the library should be allowed to call a function in the main program.

Because it is difficult to determine control flow within ELF sections or which data should be accessible by which code from binary code alone, more advanced tools are under development.

6 Prototype

6.1 Overview

We have implemented a software ELFbac prototype on AMD64 Linux, although we designed it to be readily portable across operating systems and architectures. To achieve this portability, we have created wrappers for kernel-specific primitives so that only a single header file needs to be changed to port our system to a new kernel or architecture. Furthermore, we believe our prototype could be easily adapted to other conceptually similar executable/linker formats like Microsoft's PE or Apple's Mach O.

Our implementation is based on existing virtual memory capabilities in Intel processors. We create a separate virtual memory context for each policy state so that memory accesses relevant to our policy cause page faults which we intercept in the kernel. Just as threading adds a one-to-many relationship of memory contexts to execution flows, we create a many-to-many relationship—since each execution flow can now change its memory context with each FSM state transition.

[10] As will be discussed in Section 8.2, implementing the HEAP command shown in this example will require additional work with the dynamic allocator, which is underway.

[11] System call restrictions were not used in our results section, as we are currently developing mechanisms that are portable between architectures and do not impact the existing kernel design.

[12] Every library should have its own state, and if the user specifies this, such a policy can be created. However, some libraries (such as pthreads, dl and libc) interact in ways not obvious from ELF symbols.

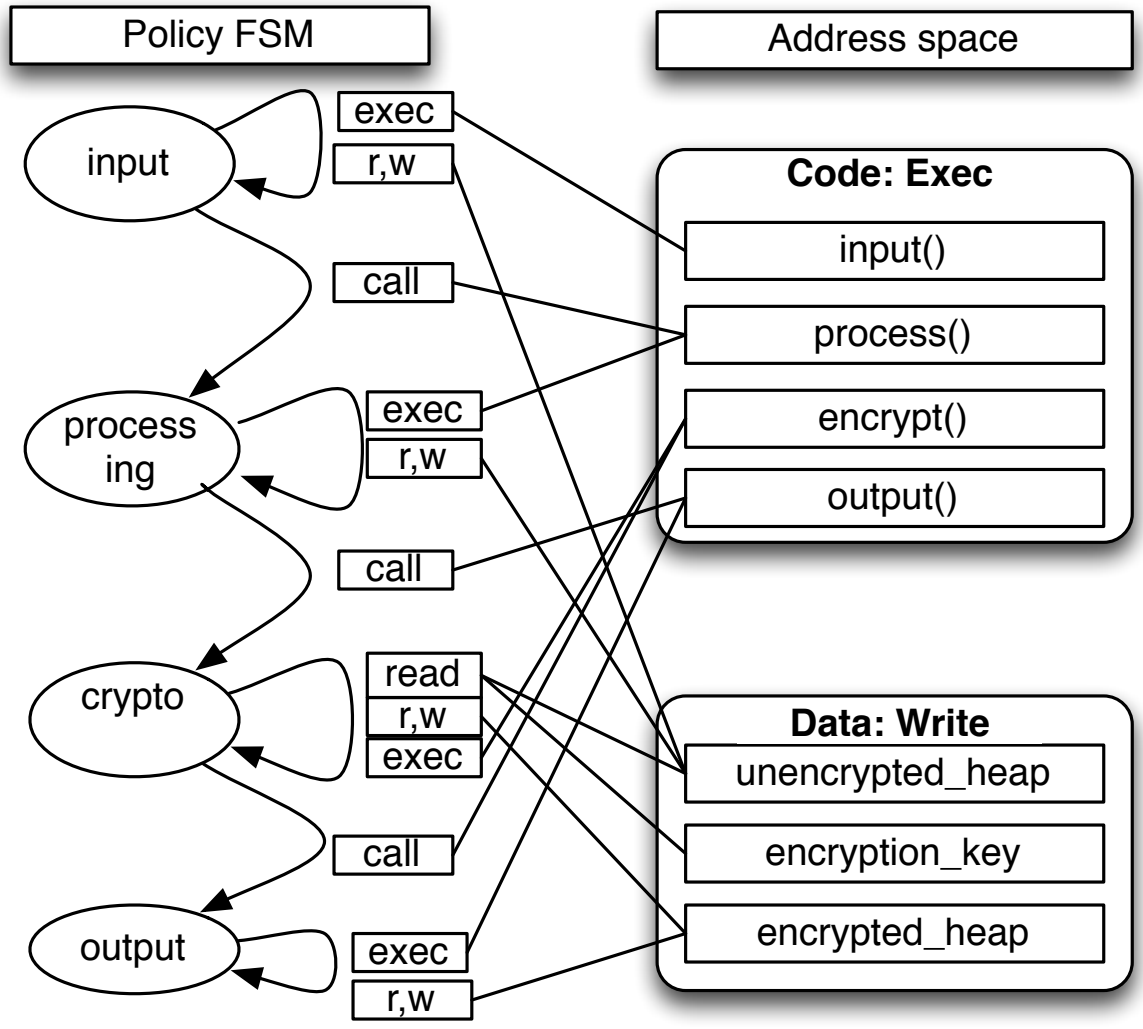

Figure 4: A sample ELFbac policy for our program example from Figure 2.

```
HEAP unencrypted_heap, encrypted_heap
input_phase exec input
input_phase read,write unencrypted_heap
input_phase syscalls * //Allow all syscalls
input_phase -> processing_phase call process

processing_phase read,write unencrypted_heap
processing_phase exec process
processing_phase -> crypto_phase call encrypt

crypto_phase exec encrypt
crypto_phase read .encryption_key
crypto_phase read unencrypted_heap
crypto_phase read,write encrypted_heap
crypto_phase -> output_phase call output

output_phase exec output
output_phase syscalls *
output_phase read encrypted_heap
```

Figure 5: Source code of the sample ELFbac policy.

6.2 Implementation

Per-state shadow memory contexts. Our prototype replaces the kernel's view of a process' virtual memory context with a collection of "shadow" contexts, one per each policy FSM state. Each shadow context only maps those regions of memory which can be accessed in the current state without violating the policy (which corresponds to loops in FSM). Any other access is trapped; the trap either causes a transition to another allowed FSM policy state, or is a policy violation.[13] On a legitimate state transition, the virtual memory context is changed; on a violation, appropriate handling logic (e.g., logging of the violation event and process termination) is invoked.

These state-specific memory contexts are only used by the page fault handler and in context switching routines. For the rest of the kernel functionality, the original virtual memory context still serves as a global view of the process address space (of which each of the shadow contexts makes a subset accessible). Therefore, we do not need to change any other subsystems in the kernel or any userspace code, as they can continue to use the system calls and kernel APIs to modify the address space, without impacting ELFbac enforcement.

The state-specific shadow contexts are lazily filled—every state starts empty and is filled by the page fault handler. Initially, an empty shadow virtual memory context is created for every state in the policy and the context corresponding to the initial state is activated. Whenever the process accesses an address not mapped in this context, the MMU raises a page fault, which ELFbac intercepts. If the access is allowed by the policy and does not result in a state transition, the corresponding memory mappings (and page table entries) are copied to the shadow context, so they will not cause a fault when accessed in the future from the same state. If, on the other hand, the access results in a state transition, the shadow virtual memory context corresponding to the new state is activated.

Whenever a region in the original memory context is changed, it is unmapped in all the shadow context, so the next access to this area will again be validated against the policy.

Additions to the Process Descriptor. Our implementation adds only three pointers to the Linux kernel task structure (`task_struct`).

[13] This implementation makes it hard to implement "once-only" access of a memory region as a policy primitive; see Section 8 for discussion of future work.

First, a pointer (`elf_policy_mm`) to a per-state separate virtual memory context is added. As explained above, this pointer is only used by the page fault handler and in context switching routines, so that the remaining large portion of the kernel, which accesses the traditional virtual memory context pointer (`tsk->mm` in Linux) to manipulate mappings, address holes or for accounting purposes, retains the illusion of a single address space per process. This reference also makes sure that the same address is never re-used and allows familiar debugging tools to work unmodified with the `ptrace` system call.

Secondly, a pointer (`elfp_current`) to the policy currently in effect for a process is stored. The policy is stored as a digraph of states, where each node stores a binary search tree of edges to other states.

Thirdly, a stack of userspace stack pointers is stored in the `elfp_stack` pointer, which is used to implement the stack-changing mechanism described in 8.2.

Per-state shadow spaces as a new caching layer. In theory, the policy is verified on every memory access by hooking the page fault handler. As trapping every memory access would be catastrophic for performance, we add a layer of "caching" on top of the existing hardware caching layers.

Consider a memory access policy in standard UNIX. The authoritative view of memory is given by the `mm_struct` struct of a process. The actual page tables (pointed to by `CR3` on x86) serve as a "just-in-time compiled" expression of the policy, since the appropriate PDE/PTE entries are created as needed by `mmap` and other system calls. Thus, the page table serves as a "caching layer" for the actual policy expressed in `vma_structs`. Moreover, page table look-ups are cached by the TLB structures (separate for instruction and data paths on x86).

We generalize this architecture by adding another layer of shadow memory for each process, within which different code sections (corresponding to the phases of the process or policy FSM states) receive different views of memory via different page tables—and must therefore generalize the caching accordingly. Namely, the shadow contexts create an extra load on the TLBs, as it would cause them to be flushed on every FSM state change. To somewhat reduce this impact, we use our "lazy" design for filling per-state shadow contexts, in which each policy state starts with an empty page table and gets the appropriate mappings copied into each on each valid access. This design emulates the relationship of between the page tables and the TLB: just like the TLB, the policy state's page table accumulates successfully accessed mappings.

As soon as the addresses are unmapped in the global context, they are deleted from each page table as well. The page table associated with the current state is loaded on each context switch.

On newer architectures, we additionally reduce the performance impact of shadow contexts by using PCID tags. As an architectural desiderata, we posit that since caching is heavily involved on the path of enforcing a memory access policy, it should in fact become an actor in enforcing this policy. This means, in particular, that cache entry invalidation must be elevated from the status of a heuristic to that of a policy primitive.

Per-state syscall checking. In addition to the above virtual memory maniputation, the current state—i.e. the label of the code unit (ELFbac section) currently executing—is verified on every syscall. Each policy state either allows or disallows each syscall; more fine-grained checking can be delegated to userspace wrapper functions.

Policy entry points and loading. All Unix-family processes (after `init`) are originally created through a `fork()`-type syscall. This syscall preserves the ELF policy of the parent process.[14] When a new executable is loaded with the `exec()` call, the policy is replaced with the policy of the new executable.

[14] If a fork should be treated as a special policy event, the `fork()` library function can be wrapped to provide the transition just after the fork. Access to the `fork()` system call from any other policy state should then be prohibited.

On dynamic binaries, we use the linker to set up the policy. If the ELF binary is dynamically linked, the kernel first loads a statically linked interpreter binary[15], whose policy will be loaded initially. The main executable is then relocated and set up by the interpreter. Since the ELFbac policy is just another ELF object, the linker is already equipped to relocate the policy as needed. Before control transfers to the main program, the interpreter calls a special system call that loads the relocated policy. Thereafter, this system call is disabled until a new binary is set up. Hence, neither the policy parser nor any other part of the kernel have to worry about how the address space was set up, e.g. by ASLR.

7 Evaluation

7.1 Need for Performance Evaluation

There is a difficulty in estimating the performance impact of ELFbac, which is inherent in the measurement of any security primitive: it ultimately depends on the software it is used to protect how often and on what code paths it is going to be hit. Our hypothesis is that well-designed software already exhibits sufficient modularity and locality to make frequent (and costly) ELFbac context switching unnecessary, and amortizing the cost of these switches over the productive sections and states of the program.

We are currently evaluating the architectures of several popular Linux daemons for the study of ELFbac policies. Since we believe the performance impact of ELFbac policies depends on the design of the software and the granularity of the specific policy, we defer performance evaluations until we can enforce a policy on a piece of well-modularized software such as the *Nginx* web server.

7.2 Security: implicit flows eliminated

The primary advantage of protecting systems with ELFbac is lessening the impact of an attacker. By preserving and enforcing intent-level semantics, an attacker who has gained code execution or an information leak in one part of the application can no longer compromise other parts. Among other benefits, ELFbac makes attack techniques like return-oriented programming much more difficult.

In order to assess the efficacy of our system, we sampled four recent vulnerabilities from the Common Vulnerabilities and Exposures database[16] that could result in code execution [17]. Because either the software was not available in the first place or adequate vulnerability details were not disclosed, we did not attempt to write an actual ELFbac policy for the relevant software. Instead we consider how a sensible default ELFbac policy, such as generated by our preliminary tools, would limit the exposure from each vulnerability.

CVE-2012-3455 In this vulnerability, a heap buffer overflow in the Word processor permits an attacker to craft evil input which subverts the main KOffice process.

ELFbac could be used to separate the Word parser and the main KOffice process. If an attacker crafts a document that can perform computation with the parser, he will no longer be able to read arbitrary files or spawn a shell, but limited to using the internal KOffice API.

CVE-2012-3639, CVE-2012-2334 In CVE-2012-3639, a bug in WebKit allows an adversary to subvert an entire browser; in CVE-2012-2334, a bug in an image library allows an adversary to subvert an entire

[15] On Linux, this is usually `/lib/ld.so`.
[16] http://cve.mitre.org
[17] http://cvedetails.com

office suite.

In both vulnerabilities, an application uses a parser for display formats (HTML and PNG, respectively). However, both parsers only require minimal interaction with the remaining components of a program. Other browsers such as Google's Chrome [5] already demonstrate that WebKit only requires a handful of calls to render bitmaps and consume input events. A PNG parser should only be able to read an input buffer and write an output buffer. Neither require access to any system calls or other program data structures.

ELFbac could stop both vulnerabilities by confining the parser to exactly what it needs to touch, thus preventing a compromised parser from subverting the rest of the application. As opposed to the Chrome security model, ELFbac does not require re-engineering of existing code and the same policy mechanism can be ported across different operating systems.

CVE-2012-0199 In this vulnerability, a bug in the processing of SQL database queries lets the attacker subvert the rest of the application.

Usually, SQL injection attacks result from input text fragments being copied into a SQL query without proper escaping. To mitigate this, various database frameworks, such as Microsoft .NET, use dedicated query builder mechanisms. With ELFbac we can assure that no code but the query builder can send data to the SQL client API, defending both against inadvertent SQL injection vulnerabilities and against the SQL layer being used after another exploitation.

8 Next Steps

8.1 Code Improvements

Our initial next steps include some improvements to the basic code.

Right now, our trapping mechanism implementation does not allow once-only accesses, although these could be implemented on the existing MMU with debugging mechanisms.

Similarly, we are working actively to implement system call restrictions in a portable and flexible manner, so untrusted code cannot perform IO operations by directly accessing kernel functions.

Furthermore, we are actively working on creating better policy tools and languages. We envision a system that analyzes binary code, programmer annotations and header files or source code if they are available, combines them with limited programmer annotations and creates a policy, which can then be manually extended for security relevant portions of a program.

8.2 Design Extensions

Improving Stack Changing ELFbac permits different FSM states to use different stacks, since stacks store both activation records and local variables which contain security-critical information.

Due to limitations in our implementation, we can only trap on a page-level granularity. Therefore, the entire stack, which usually is only one or two pages large, will be treated as one policy object. Even if we had more granular trapping, the stack's layout is changing with every function call and it would at least be severely inconvenient to keep track of current stack semantics in the kernel-level enforcement logic.

Therefore, we allow each policy state to have a separate stack and then copy parameters and relevant

data between stacks during state transitions. This cost is acceptable, because parameter records are usually short and modern ABIs like AMD64 are optimized to pass many parameters in registers.

While the current prototype requires the user to manually specify the number of parameter and return bytes to be copied between stacks, we believe this information could be deduced automatically by the policy tool. The first approach is to analyze metadata left by the compiler (e.g. in the DWARF format [13]) that specifies parameter information. Some other languages, like C++, encode parameter details in the function signature. If this information is not present or incomplete, the last instructions before the return instruction in a function or the assembly instructions immediately before a call could be analyzed.

The second approach would be to analyze header files with already accessible tools like CTAGS to find parameter information; this would be especially useful for libraries, where the header files are usually available to the programmer even if the source code is not.

However, a big issue with all these approaches is that some functions—e.g. `printf()`—consume a variable number of arguments. In practice, though, variable arguments are often used to implement some variation of string formatting, where tools already exist (for example, in the `gcc -Wformat` feature) to deduce the required number of parameters from the call site.

In any case, we could always copy a "safe" upper bound of parameters to the called stack. It might be safe to assume in a given program that no function has more than 256 bytes of parameters, so we just copy that much data if we cannot deduce specifics. This might leak sensitive stack information to called libraries and thereby reduce the added protection offered by ELFbac, but the called function at least will not be able to overwrite this data, because it only has access to a copy of the data.

Better Heap Security Dynamically allocated memory also needs to have different semantics based on what data it contains. Therefore, we will implement multiple heap areas which can be treated differently by an ELFbac policy and add a mechanism to the allocator that decides from which heap to serve an allocation request and thereby how that memory should be treated.

Another instance of a memory area that changes semantics dynamically is the heap. The heap contains both data and meta-data used by the dynamic allocator. Usually, a runtime only has a single heap that contains all objects dynamically allocated by every piece of code.

Just as with the stack, we change the runtime to include multiple heaps in disjoint memory. On modern 64-bit systems, the virtual memory range is large enough to accommodate a distinct 4 Gigabyte heap for 2^{16} security labels, but on 32-bit systems, the number of semantically different heaps is much more limited and every heap needs to be manually sized.

A modern kernel exposes two dynamic allocation schemes: the `brk()` and `mmap()` system calls. In Linux, `brk()` is implemented very similarly to `mmap()`, but with fewer parameters, so the same concepts apply. The `mmap()` syscall knows two modes of operation, which have subtly different semantics for our policy. The first mode (probably more familiar to a programmer) maps the contents of a file into a region of memory. An alternative view of this operation is to allocate a new object in memory whose contents are pre-initialized (and optionally automatically saved by the kernel periodically). The second mode (the anonymous map) allocates a region of memory (typically backed by swap, not a named file, hence "anonymous") into the memory space and is used by modern allocators to satisfy large allocations and fill their internal pools.

Both of these modes introduce new memory objects (which should have semantics) into the address space. As our policy is defined in terms of memory ranges, the allocators need to return an address in the correct range depending on the semantics. Therefore, a separate policy for `malloc()` and `mmap()` has to be introduced. The easiest way to express this policy is to rewrite program source and pass an explicit label to every `malloc()` call. As this is hugely impractical with legacy applications, we propose multiple ways of dynamically labeling legacy allocations.

The first such way is to give every policy state a default security label. For example, in the input stage of our example, all allocations could be sent to a "parser-token" heap, which would be readable by the process stage. The second approach is to tag allocations based on size. A sizable portion of dynamic allocations allocate a fixed data structure or class. Different object types tend to have different sizes, so we can deduce the type of data (and thereby its intent) from the allocation size. In cases where this does not apply (i.e. buffers of multiple elements or of varying length), we need to revert to source level analysis and rewrite each call (for example, creating a new wrapper symbol like `malloc_packet_buffer` that tags `malloc` appropriately).

8.3 Extensions

Userspace Implementation Because our implementation is rather loosely coupled to the kernel and its memory system, it is possible to implement ELFbac in the userspace as well—if one is willing to tolerate reduced security and performance. Instead of manipulating paging structures and handlers directly, one can just as well use the `mprotect()` system call and install a handler for `SIGSEGV` (in the same manner as the User Mode Linux[18] uses SIGSEGV to emulate MMU memory traps). The obvious security concern is that an attacker who can get any segment to perform a mmap() or mprotect() system call can completely circumvent the enforcing logic.

FlexTrap In our initial prototype, we will make creative use of the existing x86 virtual memory system to trap on fairly complicated conditions. However, this implementation will suffer in performance because the hardware features are rather tailored to their legacy usage patterns (some going as far back as 386). Also, the existing system offers little flexibility for security policies, being originally designed for the narrow OS-specific purpose of virtual memory support.

We envision a new x86 memory subsystem with a microcoded trapping system we call *FlexTrap*. Currently, most processors have one or more TLBs, each of which associates a page of virtual memory with a physical address. Because the TLB is minuscule compared to the size of the virtual memory space, the TLB is filled on demand from a page table, either with software routines (on SPARC and MIPS) or with dedicated hardware (Intel). However, every TLB line contains information about exactly one page of memory (modern Intel processors have two TLBs, one for small 4Kb pages and one for larger pages).

We suggest to replace this fixed-width TLB with a variable-granularity cache (in short, vTLB). Whenever the processor fails to find a TLB entry for a given virtual address mode, it switches to a special "FlexTrap" mode of operation, in which it executes regular x86 instructions starting from an address specified in a control register (similar to the page fault handler) with a separate register set, but does not perform address translation. The FlexTrap handler uses physical memory and the virtual address to compute a new entry to be placed in the vTLB and returns from the fault.

With appropriate optimization, we believe performance nearly equal to the current hardware/microcode page table handler is possible (as page table lookups are likely to be memory-bound instead of compute bound).

In the modern platform, the TLB is no longer an obscure aid to OS virtual memory translation; rather, it has become and must be treated as a security policy object. In fact, TLBs have been used for attack purposes (e.g., ShadowWalker [24]) and some defensive schemes(e.g. PaX[26]); We need to level the playing field.

[18] `http://user-mode-linux.sourceforge.net/`

9 Conclusion

Prior work in application security has put much emphasis on restricting a program's access to a system resources (such as files), and system calls. We have argued that this should be taken a step further to specify and enforce intent-level semantics of different units of code, restricting how they interact with each other. By reusing existing ELF infrastructure, we maintain compatibility with legacy code and binaries, which we have shown to be vulnerable because code units can violate their intent. Thereby we have noticeably raised the barrier to successful exploitation.

Acknowledgments

This work was supported in part by the Intel Lab's University Research Office and by the TCIPG project from the DOE (under grant DE-OE0000097). Views expressed are of the authors alone.

We would like to particularly thank Felix 'FX' Lindner and other researchers at Recurity Labs for many productive discussions and critical insights. We gratefully acknowledge the design inspirations offered by the Grsec/PaX project and the PaX team. On the subject of the ELF structure and ABI composition mechanisms, we owe many insights to the ERESI project by Julien Vanegue and the ERESI team, as well as to the Grugq's papers on the subject, and to the works by Silvio Cesare, Klog, and others published in *Phrack*; we thank Rodrigo Branco for many helpful discussions of Linux kernel security.

We would also like to thank Meredith L. Patterson of Upstanding Hackers LLC., a co-inventor of Language-theoretic Security research, for many productive collaborations.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Anonymous author. Once upon a free(). Phrack 57:9. `http://phrack.org/issues.html?issue=57&id=9`.

[3] P. Argyroudis and C. Karamitas. Heap Exploitation Abstraction by Example. OWASP AppSec Research, August 2012.

[4] A. Banerji, J. M. Tracey, and D. L. Cohn. Protected shared libraries: a new approach to modularity and sharing. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 1997. USENIX Association.

[5] A. Barth, C. Jackson, and C. Reis. The security architecture of the chromium browser. Technical report, Stanford, 2008.

[6] S. Bratus, N. D'Cunha, E. Sparks, and S. Smith. TOCTOU, Traps, and Trusted Computing. In *Proceedings of the TRUST 2008 Conference*, March 2008. Villach, Austria.

[7] S. Bratus, M. Locasto, and B. Schulte. Segslice: Towards a new class of secure programming primitives for trustworthy platforms. In A. Acquisti, S. Smith, and A.-R. Sadeghi, editors, *Trust and Trustworthy Computing*, volume 6101 of *Lecture Notes in Computer Science*, pages 228–245. Springer Berlin / Heidelberg, 2010.

[8] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. New Directions for Hardware-assisted Trusted Computing Policies (Position Paper). In D. Gawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, editors, *Future of Trust in Computing*, pages 30–37. Vieweg+Teubner, 2009.

[9] S. Bratus, M. E. Locasto, and B. Schulte. SegSlice: Towards a New Class of Secure Programming Primitives for Trustworthy Platforms. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, TRUST'10, pages 228–245, 2010.

[10] F. P. Brooks. *The Mythical Man-Month.* Addison-Wesley, 1975, 1995.

[11] S. Cesare. Shared Library Call Redirection via ELF PLT Infection. Phrack 56:7. `http://phrack.org/issues.html?issue=56&id=7`.

[12] T.-c. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *SIGOPS Oper. Syst. Rev.*, 33(5):140–153, 1999.

[13] DWARF Standards committee. *DWARF Version 4 Standard.* `http://http://dwarfstd.org/doc/DWARF4.pdf`.

[14] jp. Advanced Doug Lea's malloc exploits. Phrack 61:6. `http://phrack.org/issues.html?issue=61&id=6`.

[15] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[16] P. Loscocco and S. D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[17] J. Mauro and R. McDougall. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture.* Pearson Education, 2006.

[18] MaXX. Vudo malloc tricks. Phrack 57:8. `http://phrack.org/issues.html?issue=57&id=8`.

[19] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328. ACM, 2008.

[20] T. Müller. ASLR smack and laugh reference. In *Seminar on Advanced Exploitation Techniques.* RWTH Aachen, February 2007.

[21] Nergal. Advanced return-into-lib(c) exploits: the PaX case study. Phrack 58:4. `http://phrack.org/issues.html?issue=58&id=4`.

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

[23] J. M. B. Rivas. Overwriting the .dtors section. `http://packetstormsecurity.org/files/23815/dtors.txt.html`, December 2000.

[24] S. Sparks and J. Butler. "Shadow Walker": Raising The Bar For Rootkit Detection. BlackHat 2005. `http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf`.

[25] R. Strackx and F. Piessens. Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 2–13. ACM, 2012.

[26] P. team. 20 years of pax. In *Symposium sur la scurit des technologies de l'information et des communications*, 2012.

[27] The Debian Project. DSA-1571-1 openssl – predictable random number generator, may 2008. `http://www.debian.org/security/2008/dsa-1571`.

[28] the grugq. Cheating the ELF: Subversive Dynamic Linking to Libraries .

[29] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[30] E. Witchel, J. Rhee, , and K. Asanovic. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. In *20th ACM Symposium on Operating Systems Principles (SOSP-20)*, October 2005.

A ELFBac kernel structures

```
/* These structures describe an ELFbac policy in kernel memory. They
   are created at runtime from the elfp_desc structures found in the
   .elfbac section. This header file is intended to be used in all
   ELFbac ports, so per-kernel aliases from elfbac-linux.h are used.*/

struct elf_policy{
  struct elfp_state *states;
  struct elfp_stack *stacks;
  elfp_atomic_ctr_t refs; /*should be made atomic_t */
};
struct elfp_state {
  elfp_context_t *context; /* This memory context maps a subset of the
                              processes tables and is filled on demand */
  elfp_tree_root calls; /*Maps to OS tree implementation*/
  elfp_tree_root data;
  struct elfp_state *prev,*next; /* Linked list of states */
  struct elf_policy *policy; /* Policy this state belongs to */
  struct elfp_stack *stack; /* Last call transition taken */
  elfp_id_t id; /* id of this state in the policy. Used for parsing
                   policy statements */
};
struct elfp_stack {
  struct elfp_stack *prev,*next;
  elfp_id_t id;
  uintptr_t low,high;
  elfp_os_stack os;
};
struct elfp_call_transition{
  elfp_tree_node tree; /* Wraps OS rb-tree implementation*/
  struct elfp_state *from,*to;
  uintptr_t offset; /* Called address */
  short parambytes; /* bytes copied from caller to callee*/
  short returnbytes; /* bytes copied from callee to caller. <0 to
                              disallow implicit return */
};
struct elfp_data_transition {
  elfp_tree_node tree;
  struct elfp_state *from,*to;
  uintptr_t low, high;
  unsigned short type; /* READ / WRITE flags */
};
```