Poster: (SF)^{2}I - Structure Field Software Fault Isolation

Spyridoula Gravani, Zhuojia Shen, John Criswell
Department of Computer Science, University of Rochester
Email: {sgravani, zshen10, criswell}@cs.rochester.edu

***Abstract*—Commodity operating systems are self-extending, loading code at runtime to add new features. While useful, such self-extensibility allows attackers to inject kernel-level malware into the operating system kernel. Such malware threatens security system-wide and is not yet completely mitigated. This poster demonstrates our approach to provide safe extensibility of commodity operating system kernels.**

1. Introduction

Commodity operating systems employ monolithic designs in which the entire operating system kernel executes in the processor's privileged mode [1], [2], [3], [4]. To increase flexibility, operating systems can dynamically load executable modules at runtime in order to enhance their functionality [5]. Self-extensibility allows an operating system to tailor itself to a specific hardware configuration without restarting the system. However, it also poses a security threat since it allows attackers to load malicious code (malware) into the operating system kernel.

Once kernel-level malware infects the operating system kernel, it executes as part of the operating system in the processor's privileged mode. As part of the operating system kernel, this malware can corrupt kernel data structures to hide processes, files, and network connections [5] to keep itself invisible from administrators. It can also corrupt, steal, and delete application data as well as deny operating system services to legitimate applications.

We propose the use of virtual instruction set computing to address the safe extensibility of operating systems. Our system is based on Secure Virtual Architecture [6], a virtual instruction set infrastructure that enables the use of sophisticated program analysis techniques and transformations on operating system kernel code. Our approach consists of two major steps: 1) security policy generation for an operating system kernel, and 2) kernel code instrumentation to enforce the security policy at runtime.

2. Background

As Figure 1 shows, Secure Virtual Architecture (SVA) is a compiler-based virtual machine in which user-space programs and the operating system kernel are compiled to a virtual instruction set that is derived from the LLVM Intermediate Representation [7]. In SVA-based systems, software is shipped as virtual instruction set code. The SVA Virtual Machine instruments code on site to enforce security policies and then translates code to the processor's native instruction set for execution.

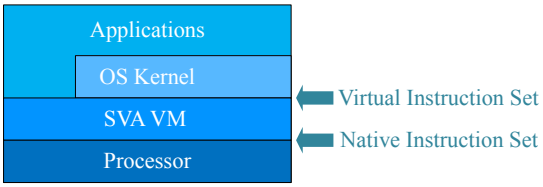


Figure 1: Secure Virtual Architecture

3. Design

We attack kernel-level malware using SVA. SVA can add run-time checks to kernel modules before loading them into the operating system kernel; our goal is to design and implement run-time checks that restrict to which structure field each kernel module can read and write. By limiting the structure fields which a kernel module can access, we limit the damage that misbehaving code can cause. For example, we can prevent modules from unlinking processes from the list of active processes [5] or changing a process's scheduling priority.

Our approach consists of two main steps: *policy creation* and *policy enforcement*. Operating systems are comprised of cooperating subsystems [3]; creating policies for each structure field that dictate which kernel modules can access that field can be tedious. We propose to classify kernel modules into categories based on their functionality and to instrument operating system kernel code so that it records information about memory accesses during execution. By collecting this information on trusted kernel code, we can automatically extract policies that can be applied to new kernel modules loaded at run-time.

Once a policy has been generated, we will instrument kernel code with run-time checks to enforce the policy, providing dynamic protection against kernel-level malware.

4. Preliminary Work

We have built an automatic access control policy inference mechanism. We developed a compiler pass that

inserts code into a user-space program that traces the loads and stores performed, compresses the traces, and extracts the access control policies. The inference mechanism will require minor extensions to support kernel code.

4.1. Tracing

We extended the SAFECode compiler [8] to instrument loads and stores to record the following metadata for each memory access: 1) the offset from the beginning of the memory object that is accessed, 2) the size of the memory access in bytes, 3) the name of the module performing the memory access, and 4) whether the access reads or writes the memory object. An application compiled with this enhanced compiler records this information during execution.

4.2. Policy Coalescing

Execution traces are large; our tools need to compact the policy generated from a trace. We observe that many fields in a memory object have a similar access pattern. We have developed a utility that coalesces the generated trace into a set of formulas that represent fields within memory objects with the same access pattern. For every memory object, we identify groups of accesses with: 1) the same access size, 2) the same access type (read or write), 3) the same module ID, and 4) the same last decimal digit of the offset in the memory object. We place these accesses into an equivalence class and create a 5-tuple of constants that represents all the memory accesses in the equivalence class.

Let S be a list of memory accesses to the same memory object that we wish to put into an equivalence class; let b denote the last decimal digit of these memory accesses. We sort the memory accesses in S by increasing offset from the beginning of the memory object. Let c be the smallest offset and d be the largest offset from the beginning of the memory object for all memory accesses in S. All offsets in the list can be represented by the formula $2x+b$, in which x is a variable that ranges from c to d with a stride equal to the difference among contiguous elements in S. If the difference among contiguous elements in S is not constant, we find the difference that occurs most frequently and remove accesses from S until all accesses are the same distance apart.

Figure 2 shows an example of coalescing. We interpret the line marked in red as follows: module M1 can read any 4-byte field in the `farray` memory object starting at an offset generated by the formula $2x+8$, where x varies from 0 to 380 with a stride equal to 20. The stride is the difference among contiguous accesses in the sorted list of offsets that have 8 as their last decimal digit.

5. Evaluation

To evaluate how well we can compress an obtained trace, we instrumented the open source tiny/turbo/throttling HTTP server [9]. We used ApacheBench to retrieve files of varying size via the loopback interface. The experiment finished in

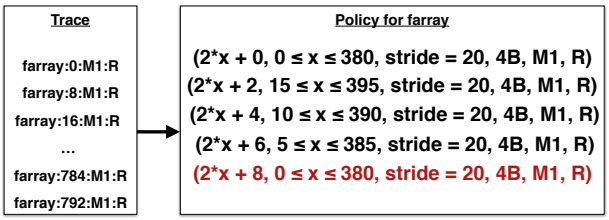
```
struct foo {
  int k;
  int l;
}
struct foo farray[100];
```

```
// Access Pattern of Module M1
int i, var;
for (i = 0; i < 100; i++) {
  var = farray[i].k;
}
```

(a) Memory object farray and access pattern of module M1

Offset	Element
0	farray[0].k
4	farray[0].l
8	farray[1].k
12	farray[1].l
..	...
792	farray[99].k
796	farray[99].l

(b) Memory Layout



(c) Coalescing

Figure 2: Example of Trace Generation and Coalescing

17 minutes; the obtained trace file required 6.6 GB of space. After coalescing, the security policy size was 338 KB.

We also collected statistics about the trace. Six distinct modules, i.e. compilation units, in the web server accessed 82 distinct memory objects. Table 1 shows the percentage of memory objects accessed by 1, 2, 3 and 4 distinct modules, respectively; no object was accessed by 5 or more modules. Our results show that most memory objects are accessed by only one module. If this pattern is common in kernel code, we can efficiently enforce strong security policies at runtime by recording the one module that needs access to the memory object; only a small number of memory objects will need more sophisticated checks to handle access by multiple modules.

TABLE 1: Web Server Memory Access Statistics

Number of Modules	% of Objects
1	75.61%
2	17.07%
3	6.10%
4	1.22%

6. Future Work and Conclusions

We leverage SVA to make self-extensibility safe for operating system kernels. Moving forward, we will enhance our compiler to recognize kernel memory allocators to apply our approach to kernel code and devise efficient methods of enforcing our per-memory object access control policies.

References

[1] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Redmond, WA, USA: Microsoft Press, 2004.

[2] A. Singh, *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[3] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Pearson Education, 2015.

[4] D. P. Bovet and M. Cesati, *Understanding the LINUX Kernel*, 3rd ed. Sebastopol, CA: O'Reilly, 2006.

[5] J. Kong, *Designing BSD Rootkits*. San Francisco, CA, USA: No Starch Press, 2007.

[6] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems," in *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, Stevenson, WA, USA, October 2007.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the Conference on Code Generation and Optimization*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[8] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[9] J. Poskanze, "thttpd - tiny/turbo/throttling http server," 2000, http://www.acme.com/software/thttpd. [Online]. Available: http://www.acme.com/software/thttpd