SEE TEXT ONLY

CADETS

Blending Tracing & Security on FreeBSD

The FreeBSD operating system has been used in many products that require strong security properties—from storage appliances, to network switches and routers as well as gaming consoles. Over the 20-plus years of FreeBSD's development, there have been significant additions to the system in the area of security.

By Jonathan Anderson, George V. Neville-Neil, Arun Thomas, and Robert N. M. Watson

The Jails [1] system introduced what today would be considered a lightweight version of virtualization. The Mandatory Access Control (MAC) framework was added in 2002 to provide fine-grained, system-wide control over what users and programs could do within the system [2]. The Audit subsystem added the ability to track—on a system-call-by-system-call basis—what actions were occurring on the system and who or what was initiating those actions [3]. More recently Capsicum has introduced capabilities into the operating system that are the basis for application sandboxes [4].

As part of a new research project, the Causal Adaptive Distributed, and Efficient Tracing System (CADETS), we are adding new security primitives to FreeBSD as well as leveraging existing primitives to produce an operating system with the maximum amount of transparency. Apart from the security implications, having better visibility into systems will help us to improve overall performance and provide new runtime debugging tools.

In this article, we'll cover our work with DTrace and the Audit system, which form the core components of the work, and talk about the challenges of using DTrace as an **always on** tracing system to track security and other events.

Starting at the Beginning

One of the main components of FreeBSD we are exploiting in CADETS is DTrace. Originally developed for Sun's Solaris operating system in the early years of the 21st century [5], DTrace was meant to solve the following problem: most software systems have some sort of logging framework, which usually depends on the ubiquitous `printf` function and formats text and sends it to some form of console.

```
printf("Hello world!");
```

All C programmers know the code shown above and every programmer knows the equivalent idiom in their own language, whether it's Python, Rust, Go, or PHP. There are a few problems with using print statements for logging systems. The first problem is that print statements have a high performance overhead. If you've ever wondered about how much work is done on the programmer's behalf by `printf` then see Brooks Davis's "Everything you ever wanted to know about 'hello, world.'" [6] Using print statements for a logging system in code that is supposed to have, otherwise, low overhead, is not an option, and so most logging systems are enabled or disabled at either compile time, using an `#ifdef/#endif` statement, or at runtime, by wrapping the logging in an `if` statement. An example of this idiom in the C language is shown below.

```
#ifdef LOGGING
if (log)
      printf("You have written %d bytes", len);
#endif /* LOGGING */
```

The next problem with print-based logging systems is that they are both static and prone to errors. If the programmer did not expose a piece of information via the logging system before the code was built, then it will not be possible, without modifying the code, to learn about any other data upon which the same function might be operating.

Together, these two problems mean that most high-performance systems, such as operating systems, are shipped without logging enabled, and when logging is enabled, the data that is available is limited to whatever the original programmer wished to expose.

As open-source developers, we are used to the idea that we can "just recompile the code," but in production systems, that's not always possible. Imagine you have sold a system to a large bank. At 3 a.m., the system has a fault of some sort and logs an error. Someone in the IT department gets a message reporting the fault, they call support, support calls a programmer, the programmer then says, "Stop the system, rebuild the code, and rerun it with logging turned on." Handling errors in that way is terrible both for the customer and for the developer. The customer will be annoyed at having to rebuild and restart their system, and the developer is unlikely to get helpful information because the error is now far in the past. Enter Dynamic Tracing.

DTrace was designed to always be available for use, without reducing system performance and without running the risk of outright crashing the system. The clearest way to think about DTrace is as a runtime debugger with significant logging and statistical capabilities. DTrace avoids the overhead of `printf` based logging systems by using a

few tricks to subvert the execution of compiled code. The complete details are covered in *The Design and Implementation of the FreeBSD Operating System* [7], but, briefly, DTrace can override the entry or exit point of any function that is compiled into the kernel or into a user-space program. The way in which functions are collected into libraries and programs is a well-defined process, and each function has a set of instructions that indicate where the function begins. When DTrace traces a function, it replaces a few instructions with some of its own so that when the code reaches that point, DTrace's code gets called first, and it can collect and process the function's argument. The practical upshot of this is that when DTrace is not in use, it has zero overhead.

Tracing for Security

How can we apply DTrace to the problems of security? One aspect of security is finding where the bad actor lies in a system. A key question to ask when looking at a system is "Who did what to whom and when?" We'll refer to this as "Question 1." Establishing the tree of operations such that we can trace it back to its root is one part of forensic analysis that can help us find our bad actor and also show us what we need to change to prevent future security breaches.

Imagine that, regardless of the performance cost, we had complete transparency into every operation performed on a computer system. With sufficient time, analysis, and tooling, we would be able to take the output generated by the tracing system and find out the answer to Question 1.

DTrace gives us the basis on which to build such a system, but there remain some challenges. In the previous section, we stated that using some clever tricks to replace certain instructions at runtime, DTrace would have zero overhead **when not in use**. That feature of DTrace was what made it viable to ship it, by default, with Solaris, and the FreeBSD and Mac OS in the first place. It was not until there was a way to ship a tracing system that didn't have a performance impact on a running system that such a system could be fielded. What happens when DTrace begins tracing? Depending on what is being traced and how much data is being collected, the overall performance of the system will be impacted to a greater or lesser degree. If the overhead introduced by tracing gets too high, then the kernel will terminate the tracing as a form of self-preservation. The second tenet of the DTrace system, that the tracing system must not unduly tax the system, is one of the key challenges of using DTrace as a security technology. An attacker that knows there might be tracing will first cause there to be a great deal of irrelevant load on the system, causing the tracing system to exit, and then they will go about attacking the system. Another component of the CADETS project looks at the provenance of traces in order to thwart such attacks on the tracing system itself.

The majority of the current use cases for DTrace involve using it as a runtime debugger, turning on tracing when someone suspects there is a problem on a system, rather than leaving the tracing running all the time. A small subset of users have built complex telemetry systems around DTrace, including Fishworks [8], but in these first attempts at **always-on tracing** the number of things being traced was a small subset of the possible tracepoints. To build a system that has complete visibility, we need to not only have always-on tracing, but to increase the performance of the tracing system such that the overhead collecting the data does not overwhelm the system's ability to do productive work. Another requirement of a tracing system targeted at security is that trace records cannot be dropped or lost due to high load. DTrace was designed in such a way that under high load it was acceptable to drop trace records, long before the kernel might terminate the `dtrace` collection process due to the system being unresponsive. A system that is trying to collect trace data for later forensic analysis turns that concept on its head.

Any system where tracing is always on will generate a lot of data, and that data will need to be analyzed to track down attackers and how they are able to compromise the system. There are several systems for taking arbitrary textual output from various tools and trying to make some sense of it, including Splunk [9]. The goal of the CADETS project is to produce data for use by such tools. It will be much easier for consumers if the data is in a machine-readable format.

Trace Records for Software Tools

As we began our work using DTrace as an

always-on tracing system, we quickly realized that parsing the voluminous output generated by the system would present a problem not only for human analysts, but also for any tools that would be consuming the data. One of the first features we added to DTrace for CADETS was **machine-readable output**. Using the `libxo` library, we converted DTrace to not only produce plain text, but also XML, JSON, and HTML, the three output formats supported by `libxo`. At the time that we added machine-readable output, the Illumos version of DTrace did have a way of outputting JSON, but this was via a print-like operation rather than a pervasive change. In the CADETS version of DTrace, a single command-line option changes all the output

```
```
dtrace -n 'syscall::write:entry'
dtrace: description 'syscall::write:entry' matched 2 probes
CPU   ID   FUNCTION:NAME
      0 59780   write:entry
      0 59780   write:entry
Example 1```

```
# dtrace -O json -n 'syscall::write:entry'
dtrace: description 'syscall::write:entry' matched 2 probes
CPU     ID                    FUNCTION:NAME
{
     "probe": {
        "timestamp": 3594774042481656,
        "cpu": 1,
        "id": 59780,
        "func": "write",
        "name": "entry"
 }
Example 2
```

the user sees from DTrace from plain text into a machine-readable format.

The code samples above show the differing output between the plain, textual output, and machine-readable output. Our example asks DTrace to trace all calls into the `write` system call. Example 1 shows the default, textual output from the `dtrace` command, which is arranged in columns. To write a tool that parses the output requires knowing quite a bit about the output, because the columns are only labeled at the start of the output. Example 2 shows the same tracepoint, but with the addition of the `-O json` command line argument, instructing `dtrace` to give us all output in JSON format. The machine-readable output tags each element, starting with the `probe`, which is an object that contains several elements, including the `cpu`, `id`, `func` and `name`, all of which appeared, untagged, in the machine-unreadable output of Example 1. One addition for machine-readable output is the timestamp element, which reports the time that the probe fired, in nanoseconds, since the UNIX epoch. While a DTrace script can output the time using either the `timestamp` or `walltimestamp` variables, we believed that having a `timestamp` in every machine-readable record would simplify building tools for security forensics. While probes may fire in parallel on different cores, indicated by the `cpu` variable, modern Intel-based systems have a synchronized timestamp, which DTrace uses, meaning that the time stamps are an excellent indication of the order of operations on a single system.

DTrace and Audit

The `audit` subsystem has been a part of FreeBSD since 2004 and is an optional kernel component, along with a user space daemon `auditd`, which implements a "fine-grained, configurable logging of security related events [10]. It was built to meet the "Common Criteria (CC) Common Access Protection Profile (CAPP) evaluation," a security standard set out by the U.S. Government [11]. The audit subsystem adds a set of handcrafted tracepoints via C macros to parts of the kernel where access to data and resources take place. For example, in a system with `audit` enabled, any time a file descriptor is accessed, a note is made in an audit record. Audit records are periodically flushed to permanent storage.

Our recent work with DTrace and security led us to desire a bridge between the `audit` system and DTrace. Robert Watson added an `audit provider` to the DTrace system in FreeBSD. A

DTrace provider gives access to a set of tracepoints from within DTrace. Some well-known examples of providers are those dealing with Function Boundary Tracepoints (fbt), System Calls (syscall), and network protocols (tcp, udp, ip). The audit provider gives DTrace the ability to record information about audit events that occur on the system while also applying DTrace features, such as filtering events via predicates and collecting statistical information via aggregations.

Using the audit system in the absence of DTrace, we did not have a convenient way to write runtime analysis scripts that allowed us to more finely target the processes that we wanted to investigate. The audit system will target a particular process, but we wanted to be able to collect data only when that process took a particular action.

Consider a scenario where we want to see who is talking to a web server; we might decide that we only want to know about connections that are coming from a specific set of Internet addresses, perhaps because we know those addresses have already been identified as part of a botnet. With the Audit Provider we can write a simple D script that asks only for the audit events relating to connect(2), and then filter those events based on the IP addresses they contain. Performing this data reduction as close to the source of the information as possible not only reduces the overall load on the system, but it also reduces the amount of data a human analyst or software tool has to look at during the later analysis phase.

OpenDTrace: The Future of DTrace

Besides Illumos, two other operating systems projects have ported and adopted DTrace. FreeBSD has had a port of DTrace since 2008 and Apple's Mac OS since 2007, where it is also integrated into the `Instruments` performance-analysis tool. With the demise of Sun Microsystems in 2010, the development of DTrace split, with some work being done within Oracle, but much of it moved onto Illumos, the fully open-source follow-on to OpenSolaris. Both FreeBSD and Mac OS continued to bring in changes from Illumos, but these were, for the most part, bug fixes rather than large new features. All three groups have done what they could to share code among themselves, but there

are significant differences among all three kernels. DTrace, although it was written in a good, portable style, requires many specialized hooks into the operating system, and this has resulted in some level of code drift. For example, the FreeBSD version of DTrace works on both ARMv8 and ARMv7 processors, but this is not yet true on Illumos. As part of our CADETS work, we realized that we wanted to add new features to DTrace and also to further abstract the code from any particular operating system, which led us to create OpenDTrace.

The aim of OpenDTrace is to provide a single, unified upstream for DTrace code that can then be easily imported into other operating systems, including FreeBSD, Mac OS, and Illumos, as well as others. The approach is similar to that taken by OpenBSM [12] and OpenZFS [13]. With this unified code base in place, we can then add features that apply to all the downstream OS consumers of DTrace far more quickly than we do today. .

Acknowledgment

The authors thank Ripduman Sohan for comments that greatly improved the manuscript. ●

JONATHAN ANDERSON is an Assistant Professor in Memorial University of Newfoundland's Department of Electrical and Computer Engineering, where he works at the intersection of operating systems, security, and software tools such as compilers. He is a FreeBSD committer and is always looking for new graduate students with similar interests.

GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking, and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City.

ARUN THOMAS is a researcher at BAE Systems R&D and the principal investigator of the the Causal, Adaptive, Distributed, and Efficient Tracing System (CADETS) project.

DR ROBERT N. M. WATSON is a Senior Lecturer (Associate Professor) at the University of Cambridge Computer Laboratory, where he leads research spanning operating systems, security, and computer architecture. He is a FreeBSD developer, member of the FreeBSD Foundation Board of Directors, and coauthor of *The Design and Implementation of the FreeBSD Operating System* (second edition).

This work has been sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

REFERENCES

[1] "Jails: Confining the omnipotent root." Poul-Henning Kamp <phk@FreeBSD.org> Robert N. M. Watson <rwatson@FreeBSD.org> (2000)

[2] Watson, R.; Feldman, B.; Migus, A.; and Vance, C. "Design and implementation of the Trusted BSD MAC framework," Proceedings—DARPA Information Survivability Conference and Exposition, DISCEX 2003, 1(Discex Iii), 38–49. http://doi.org/10.1109/DISCEX.2003.1194871 (2003)

[3] Watson, R. N. M. and Salamon, W. "The FreeBSD Audit System," UKUUG LISA Conference, 1–6. (2006)

[4] Watson, R. N. M.; Anderson, J.; Laurie, B.; and Kennaway, K. "Capsicum: practical capabilities for UNIX," 19th Usenix Security Symposium, (Figure 1), 3. http://doi.org/10.1145/2093548.2093572 (2010).

[5] Cantril, B.; Shapiro, M. and Leventhal, A. "Dynamic Instrumentation of Production Systems." (2004)

[6] Davis, B. *Everything you ever wanted to know about "hello, world"* (*but were afraid to ask)*, BSDCan. (2016)

[7] McKusick, M.; Neville-Neil, G.; and Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System, Second Edition*. Boston, Massachusetts: Pearson Education. (2014)

[8] http://dtrace.org/blogs/bmc/2008/11/10/fishworks-now-it-can-be-told/

[9] https://www.splunk.com

[10] https://www.freebsd.org/cgi/man.cgi?query=audit&sektion=4

[11] https://www.niap-ccevs.org/pp/pp_os_ca_v1.d.pdf

[12] http://www.trustedbsd.org/openbsm.html

[13] http://open-zfs.org/wiki/Main_Page