Embedded in Academia

{ 2015 11 23 }

Multi-Version Execution Defeats a Compiler-Bug-Based Backdoor

[This piece is jointly authored by Cristian Cadar, Luís Pina, and John Regehr]

What should you do if you're worried that someone might have exploited a compiler bug to introduce a backdoor into code that you are running? One option is to find a bug-free compiler. Another is to run versions of the code produced by multiple compilers and to compare the results (of course, under the additional assumption that the same bug does not affect all the compilers). For some programs, such as those whose only effect is to produce a text file, comparing the output is easy. For others, such as servers, this is more difficult and specialized system support is required.

Today we'll look at using Varan the Unbelievable to defeat the sudo backdoor from the PoC||GTFO article. Varan is a multi-version execution system that exploits the fact that if you have some unused cores, running additional copies of a program can be cheap. Varan designates a leader process whose system call activity is recorded in a shared ring buffer, and one or more follower processes that read results out of the ring buffer instead of actually issuing system calls.

Compilers have a lot of freedom while generating code, but the sequence of system calls executed by a program represents its external behaviour and in most cases the compiler is not free to change it at all. There might be slight variations e.g., due to different compilers using different libraries, but these can be easily handled by Varan. Since all correctly compiled variants of a program should have the same external behaviour, any divergence in the sequence of system calls across versions flags a potential security attack, in which case Varan stops the program before any harm is done.

Typically, Varan runs the leader process at full speed while also recording the results of its system calls into the ring buffer. However, when used in a security-sensitive setting, Varan can designate some system calls as blocking, meaning that the leader cannot execute those syscalls until all followers have reached that same program point without diverging. For sudo, we designate `execve` as blocking, since that is a point at which sudo might perform an irrevocably bad action.

So here's the setup:

1. We have a patched version of sudo 1.8.13 from the PoC||GTFO article. It runs correctly and securely when compiled by a correct C compiler, but improperly gives away root privileges when compiled by Clang 3.3 because the patch was designed to trigger a wrong-code bug in that compiler.
2. We are going to pretend that we don't know about the Clang bug and the backdoor. We compile two versions of the patched sudo: one with Clang 3.3, the other with the default system compiler, GCC 4.8.4.
3. We run these executables under Varan. Since the critical system call `execve` is blocking, it doesn't much matter which version is the leader and which is the follower.

Now let's visit an Ubuntu 14.04 VM where both versions of sudo (setuid root, of course) and Varan are installed. We're using a user account that is not in the sudoers file — it should not be allowed to get root privileges under any circumstances. First let's make sure that a sudo that was properly compiled (using GCC) works as expected:

```
$ /home/varan/sudo-1.8.13/install/bin/sudo-gcc cat /etc/shadow
Password:
test is not in the sudoers file.  This incident will be reported.
```

Next, we make sure that the backdoor is functioning as intended:

```
$ /home/varan/sudo-1.8.13/install/bin/sudo-clang cat /etc/shadow
Password:
root:!:16693:0:99999:7:::
...
test:$6$nL9rosCD$A.prR28WWZYgDHy.m6ovWYaRhP/OhOX7V9nDDirm9ZffjTWhLdJ7C171wTfLBpQNC2EoeEKAJpcEz9gV9ravL/:16693:0:99999:7:::
messagebus:*:16757:0:99999:7:::
colord:*:16757:0:99999:7:::
```

So far so good. Next let's try the gcc-compiled sudo as the leader with the backdoored sudo as the follower:

```
$ vx-suid /home/varan/sudo-1.8.13/install/bin/sudo-gcc \
          /home/varan/sudo-1.8.13/install/bin/sudo-clang -- cat /etc/shadow
Password:
$
test is not in the sudoers file.  This incident will be reported.
```

What happened here is that the gcc-compiled leader runs as before, since it doesn't ever try to execute an `execve` call. When the backdoored follower tries to execute the malicious `execve` call, Varan detects the divergence and terminates both processes safely.

Now let's try switching around the leader and follower, i.e., run the backdoored sudo as the leader with the gcc-compiled sudo as the follower:

```
$ vx-suid /home/varan/sudo-1.8.13/install/bin/sudo-clang \
          /home/varan/sudo-1.8.13/install/bin/sudo-gcc -- cat /etc/shadow
Password:
$
```

This time the leader tries to execute the malicious `execve` call, and Varan blocks its execution until the follower reaches the same system call or diverges. In this case, the follower tries to execute a `write` system call (to print "`test is not in the sudoers file...`") and thus Varan detects divergence and again terminates execution safely.

In this example, we only ran two versions in parallel, but Varan can run more than two versions. In terms of performance

and resource utilization, security applications like sudo are a great match for multi-version execution: they are not CPU-bound, so any performance degradation is imperceptible to the user, and the extra cores are needed only briefly, during the critical security validation checks. We are looking into applying this approach to other critical security applications (e.g. ssh-agent and password managers), and are investigating a way of hardening executables by generating a single binary with Varan and a bunch of versions, each version generated by a different compiler. We can then deploy this hardened executable instead of the original program.

Of course, Varan can detect misbehavior other than compiler-bug-based backdoors. Divergence could be caused by a memory or CPU glitch, by a plain old compiler bug that is triggered unintentionally instead of being triggered by an adversarial patch, or by a situation where an application-level undefined behavior bug has been exploited by only one of the compilers, or even where both compilers exploited the bug but not in precisely the same way. A nice thing about N-version programming at the system call level is that it won't bother us about transient divergences that do not manifest as externally visible behaviour through a system call.

We'll end by pointing out a piece of previous work along these lines: the Boeing 777 uses compiler-based and also hardware-based N-version diversity: there is a single version of the Ada avionics software that is compiled by three different compilers and then it runs on three different processors: a 486, a 68040, and an AMD 29050.

Posted by regehr on Monday, November 23, 2015, at 7:51 am. Filed under Compilers, Computer Science, Software Correctness. Follow any responses to this post with its comments RSS feed. You can post a comment, but trackbacks are closed.

{ 11 } Comments

1. Christian Vetter | November 23, 2015 at 9:05 am | Permalink

   Nice idea.

   How would you handle non-deterministic programs, though? E.g. you have a command that needs to read 20 files, and decides to do it in parallel. The order of system calls would be somewhat arbitrary, based on the CPU/IO scheduler. You could end up in a dead-lock between client and server process, couldn't you?

2. bcs | November 23, 2015 at 11:25 am | Permalink

   Re threading: if you can pick pick out thread synchronization events, you could construct a partial order of the syscalls and validate those are the same. With a post compilation variant of tsan (can you do per thread page table permissions?) to detect data races, that might be rather robust.

   IIRC there was a group playing something like that game in reverse; run a single program under a VM to allow repeatedly forcing many possible total order from the program enforced partial order and check that they all "work".

3. Alex Groce | November 23, 2015 at 12:57 pm | Permalink

   I think if you only declare a handful of critical things (e.g. execve) blocking, you should usually be ok, since the leader can continue without followers, and should in the end (for most reasonable parallelisms) run the same set of system calls. If you read n files in parallel to look for something, and terminate the search on a successful detection, though, you will end up with followers stuck. I assume if leader terminates and followers are waiting around, that might be reported but the work still gets done by the leader.

4. regehr | November 23, 2015 at 12:59 pm | Permalink

   Hi Christian, Varan (not my work, just to be clear) handles threads by giving a separate ring buffer to each thread in the leader– the paper contains some good details:
   http://srg.doc.ic.ac.uk/files/papers/varan-asplos-15.pdf
   I don't know about deadlock but I bet Cristian would explain more if we asked!

5. Alex Groce | November 23, 2015 at 1:00 pm | Permalink

   I guess it depends on what flags divergence. If you flag as soon as a thread of the follower wants to perform a syscall leader hasn't yet performed…

6. regehr | November 23, 2015 at 1:00 pm | Permalink

   bcs, I remember reading a paper like you're talking about but don't have a link or title handy.

7. regehr | November 23, 2015 at 1:01 pm | Permalink

   Alex, Cristian told me there are some thorny issues regarding termination upon divergence but I didn't get the details.

8. Alexander Monakov | November 23, 2015 at 1:51 pm | Permalink

   Note that as soon as the monitored program mmap's a file with write access, or likewise opens a shared memory segment, it is no longer possible to execute N copies of it skipping only system calls: any writes to shared storage will introduce data races, any atomic updates will be duplicated N times, etc. I've only skimmed the ASPLOS paper, but it doesn't seem to address this issue?

9. regehr | November 23, 2015 at 2:19 pm | Permalink

   Alexander, I don't know the implementation but the paper says it handles 86 system calls, all that were needed to

run the benchmark applications. I have to assume this includes mmap() since it is extremely common.

10. Peter | November 23, 2015 at 3:55 pm | Permalink

    Regarding the Boing 777: are you sure it’s just a single source code compiled by 3 compilers? Some years ago in univeristy I had a class about saftey in aircraft systems engineering and was told that the saftey critical software is written by multiple independent teams to mitigate bugs. So I would guess its 3 different sources compiled by 3 compilers running on 3 architectures…
    I couldn’t find clear info in the linked pdf.

11. regehr | November 24, 2015 at 12:34 am | Permalink

    Hi Peter, you’re right that the paper doesn’t quite come out and say this, but my understanding is they implemented the SW once.