Linux Based Inter-Process Code Injection Without Ptrace(2)

Tuesday, September 5, 2017 At 5:01AM

Using the default permission settings found in most major Linux distributions it is possible for a user to gain code injection in a process, without using ptrace. Since no syscalls are required using this method, it is possible to accomplish the code injection using a language as simple and ubiquitous as Bash. This allows execution of arbitrary native code, when only a standard Bash shell and coreutils are available. Using this technique, we will show that the noexec mount flag can be bypassed by crafting a payload which will execute a binary from memory.

AppArmor). The Linux kernel offers documentation for the different values this setting can be set to. For the purposes of this injection, there are two pairs of settings. The lower security settings, 0 and 1, allow either any process under the same uid, or just the parent process, to write to a processes /proc/${PID}/mem file, respectively. Either of these settings will allow for code injection. The more secure settings, 2 and 3, restrict writing to admin-only, or completely block access respectively. Most major operating systems were found to be configured with '1' by default, allowing only the parent of a process to write into its /proc/${PID}/mem file.

This code injection method utilises these files, and the fact that the stack of a process is stored inside a standard memory region. This can be seen by reading the maps file for a process:

```
$ grep stack /proc/self/maps
7ffd3574b000-7ffd3576c000 rw-p 00000000 00:00 0                          [stack]
```

Among other things, the stack contains the return address (on architectures that do not use a 'link register' to store the return address, such as ARM), so a function knows where to continue execution when it has completed. Often, in attacks such as buffer overflows, the stack is overwritten, and the technique known as ROP is used to assert control over the targeted process. This technique replaces the original return address with an attacker controlled return address. This will allow an attacker to call custom functions or syscalls by controlling execution flow every time the ret instruction is executed.

This code injection does not rely on any kind of buffer overflow, but we do utilise a ROP chain. Given the level of access we are granted, we can directly overwrite the stack as present in /proc/${PID}/mem.

Efforts were made to limit the external dependencies of this script, as in some very restricted environments utility binaries may not be available. The current list of dependencies are:

- GNU grep (Must support -Fao –byte-offset)
- dd (required for reading/writing to an absolute offset into a file)
- Bash (for the math and other advanced scripting features)

The general flow of this script is as follows:

Launch a copy of sleep in the background and record its process id (PID). As mentioned above, the sleep command is an ideal candidate for injection as it only executes one function for its whole life, meaning we won't end up with unexpected state when overwriting the stack. We use this process to find out which libraries are loaded when the process is instantiated.

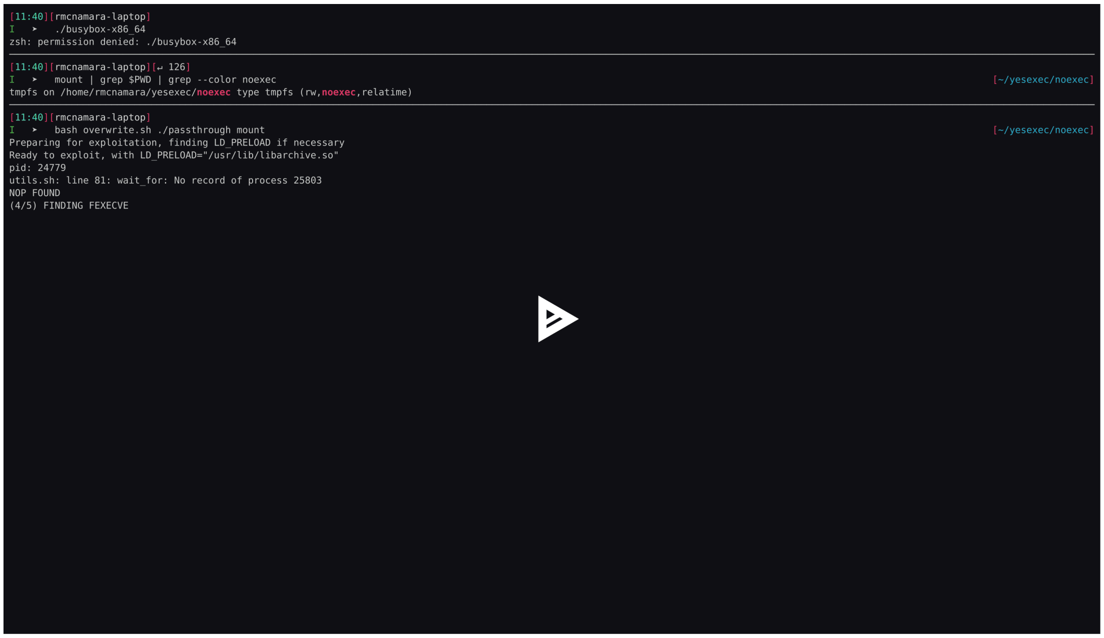
Using /proc/${PID}/maps we try to find all the gadgets we need. If we can't find a gadget in the automatically loaded libraries we will expand our search to system libraries in /usr/lib. If we then find the gadget in any other library we can load that library into our next slave using LD_PRELOAD. This will make the missing gadgets available to our payload. We also verify that the gadgets we find (using a naive 'grep') are within the .text section of the library. If they are not, there is a risk they will not be loaded in executable memory on execution, causing a crash when we try to return to the gadget. This 'preload' stage should result in a possibly empty list from our payload description file into a ROP payload. For example, for a 64bit system, the line 'syscall 60 0' will convert to ROP gadgets to load '60' into the RAX register, '0' into RDI, and a syscall gadget. This should result in 40 bytes of data: 3 addresses and 2 constants, all 8 bytes. This syscall, when executed, would call exit(0).

We can also call functions present in the PLT, which includes functions imported from external libraries, such as glibc. To locate the offsets for these functions, as they are called by pointer rather than syscall number, we need to first parse the ELF section headers in the target library to find the function offset. Once we have the offset we can relocate these as with the gadgets, and add them to our payload.

String arguments have also been handled, as we know the location of the stack in memory, so we can append strings to our payload and add pointers to them as necessary. For example, the fexecve syscall requires a char** for the arguments array. We can generate the array of pointers before injection inside our payload and upon execution the pointer on the stack to the array of pointers can be used as with a normal stack allocated char**.

Once the payload has been fully serialized, we can overwrite the stack inside the process using dd, and the offset to the stack obtained from the /proc/${PID}/maps file. To ensure that we do not encounter any permissions issues, it is necessary for the injection script to end with the 'exec dd' line, which replaces the bash process with the dd process, therefore transferring parental ownership over the sleep program from bash to dd.

A proof of concept video shows this passthrough payload allowing execution of a binary in the current directory, as a standard child of the shell.



Future work:

To speed up execution, it would be useful to cache the gadget offset from its respective ASLR base between the preload and the main run. This could be accomplished by dumping an associative array to disk using kernel.yama.ptrace_scope=2

Other mitigation strategies include combinations of Seccomp, SELinux or Apparmor to restrict the permissions on sensitive files such as /proc/${PID}/maps or /proc/${PID}/mem.

The proof of concept code, and Bash ROP generator can be found at https://github.com/GDSSecurity/Cexigua

Author: Rory McNamara

Gotham Digital Science, LLC (GDS), is a wholly owned subsidiary of Aon, PLC

This material has been prepared for informational purposes only and should not be relied on for any other purpose. You should consult with your own professional advisors or IT specialists before implementing any recommendation or following the guidance provided herein. Further, the information provided and the statements expressed are not intended to address the circumstances of any particular individual or entity. Although we endeavor to provide accurate and timely information and use sources that we consider reliable, there can be no guarantee that such information is accurate as of the date it is received or that it will continue to be accurate in the future. The examples provided in this article are not based upon an actual Stroz/Aon client, but was provided for illustrative purposes only.

About Cyber Solutions

AON

Business Email: *

Business Phone:

Job Title: *

Company: *

Location: *

--Please select--

How may we help you?

Aon and other Aon group companies will use your personal information to contact you from time to time about other products, services and events that we feel may be of interest to you. All personal information is collected and used in accordance with our privacy statement.

Please click here if you do not wish to receive these communications.

I'm not a robot

reCAPTCHA
Privacy - Terms

Submit