Moving Vulnerable Kernel Logic Into User-Space

Oakland University

Ryan Raymond

April 21, 2023

Contents

Chapter 1

Abstract

Programs which run at ring zero are a security risk. Moving code into user space can mitigate this risk. As a demonstration, a simple kernel module with vulnerable logic was used as a control case. This module was capable of targeted data exfiltration from the kernel. It was shown that moving the application logic into user space decreased the severity of the vulnerability. Furthermore, rewriting the user space program in Guile – a very high-level language – removed the vulnerability entirely.

Chapter 2

Introduction

The larger the Linux kernel's code-base grows, the more vulnerable it becomes. Because the kernel runs in ring zero, any vulnerability could be extremely dangerous. To make matters worse, the use of the C language and manual memory management drastically increases the incidence of vulnerabilities.

For example, the following are a small selection of the most severe memory corruption vulnerabilities of 2022. [1]

CVE-2022-26772 A memory corruption issue was addressed with improved state management. This issue is fixed in macOS Monterey 12.4. An application may be able to execute arbitrary code with kernel privileges.

CVE-2022-25651 Memory corruption in Bluetooth host due to integer overflow while processing BT HFP-UNIT profile in Snapdragon Auto, Snapdragon Consumer IOT, Snapdragon Industrial IOT, Snapdragon Mobile, Snapdragon Voice & Music

CVE-2021-44179 Adobe Dimension versions 3.4.3 (and earlier) is affected by a memory corruption vulnerability due to insecure handling of a malicious GIF file, potentially resulting in arbitrary code execution in the context of the current user. User interaction is required to exploit this vulnerability.

Despite these risks, large sections of convoluted code are added to the Linux Kernel regularly, including the Bluetooth stack, and device drivers, which have introduced severe vulnerabilities. It is pertinent to develop methods for moving this vulnerable code out of the kernel and into user-space, where high-level languages can be used, and vulnerabilities will be less able to affect the rest of the kernel. This technique is most advantageous in instances where the relevant code either has low performance requirements, or is especially convoluted and long.

Chapter 3

Literature

There is substantial variation between the architectures of the various major OS kernels in the market today. While Apple's Darwin and Windows NT are both hybrid kernels in which many device and network drivers are implemented in user space, the Linux kernel features a monolithic design in which all device drivers are implemented in kernel space.

There are real benefits to such a design, including improved performance, simplified design, and protection of essential functionality from interference by user space programs. However, there are also vulnerabilities inherent with this design. Because kernel code is run at ring zero, vulnerabilities are very severe. For example, BlueBorne, which exploits a vulnerability in the Linux Bluetooth stack, can perform arbitrary code execution as root [2].

Code written in C is often vulnerable to memory errors such as use-after-free, buffer overflow, or integer overflow. Many such vulnerabilities have been reported in the Linux kernel, for example, CVE-2015-3636 [3]. Using a higher-level language may decrease the incidence of such vulnerabilities. POSIX compliant kernels have been written in high-level languages like Go, and while they suffer from decreased performance, there are some situations where the security benefit might justify the performance loss [4].

For example, if the Linux Bluetooth stack had been implemented in user space, and in a high-level language, the BlueBorne vulnerability, and others like it, might have been avoided.

Chapter 4

Methodology

SStore

While it is possible to find real-world examples of vulnerabilities in Linux kernel modules, creating a vulnerable module from scratch makes both the problem and solution simpler, easier to implement, and more illustrative. One of the simplest and most difficult-to-avoid C anomalies are buffer overflows. These are common wherever manual memory management, arrays, or strings are used, and can either be used for memory corruption or exfiltration. Exfiltration is preferable to corruption for the purposes of demonstration because it reduces the chance of damaging the host system during development. Therefore, a module which allows for the exfiltration of data through buffer overflows has been developed.

”String-store” (hereafter referred to as `SStore`) is a simple kernel module for the storage of strings. The user may interact with the module using the character device `/dev/SStore0`. The user should be able to insert a string `"foobar"` into array index M by writing the following:

```
# echo "i{M}" > /dev/SStore0
# echo "{foobar}" > /dev/SStore0
```

In order to retrieve a string from index M:

```
# echo "r{M}" > /dev/SStore0
# head -n 1 > /dev/SStore0
{foobar}
```

Reading a line from `/dev/SStore` should return the string `foobar` followed by a newline character.

4.1 Monolithic

The monolithic implementation of `SStore` works exactly according to the preceding specification, with one small modification. For the sake of simplicity, the monolithic module only accepts strings up to 16 characters long, plus a null byte. Anything after the 16th character is truncated.

```
# echo "i0" > /dev/SStore0
# echo "Here is a very long string" > /dev/SStore0
# echo "r0" > /dev/SStore0
# head -n 1 /dev/SStore0
Here is a very l
```

4.2 Micro

The homogeneous implementation of `SStore` makes a slight departure from the monolithic, as instead of storing strings in kernel space, it keeps them in user-space. This was accomplished by creating two character devices instead of one. In this implementation, there is the expected kernel-like interface `/dev/SStore0`, however, there is also `/dev/SSS0` (String Store Server) which allows the kernel to control a user-space program. Two such programs were written; one in C, and another in a Lisp-like language called Guile.

Instead of storing strings in kernel space, the homogeneous module relays its instructions to the user-space server program. The module then then receives the strings from the server and returns them to the user-space interface. Again, for simplicity's sake, strings are limited in length to 16 characters.

4.3 Miscellanious

Vulnerabilities like the kind designed here may be exploited by systematically reading data until something important is found. However, in order to make the results of the experiment reproducible, several small changes were made involving memory addresses. Firstly, a small kernel module `flag.ko` was written. This module simply creates a flag string in kernel-space, and then logs its memory address. Second, those programs written in C (being the kernel module and C micro_server) were slightly modified to also log the memory addresses of the strings they printed.

These two changes should make it easy to reproduce the vulnerability. They do not affect either server's ability to be exploited in the wild.

Chapter 5

Findings

Monolith

Inserting the `flag.ko` module and reading the kernel message buffer:

```
# insmod flag.ko

$ tail -n 1 /var/log/messages

...kernel:  Secret location:  18446744072644022272
```

Loading the `monolith.ko` module, and reading string number 0 from `/dev/SStore0`:

```
# insmod monolith.ko

# echo "r0" > /dev/SStore0

# head -n 1 /dev/SStore0


$ tail -n 1 /var/log/messages
```

```
...kernel: Source: 18446744072645694592
```

Calculating the distance between index 0 and the flag:

```
$ echo $(( 18446744072644022272 - 18446744072645694592))
-1672320
```

Calculating the string index which should land on the flag, assuming each string is 17 characters long, with the last character being a null byte:

```
$ echo $((-1672320 / 17))
-98371
```

Using this offset, it is possible to exfiltrate the flag which was placed in the kernel earlier.

```
# echo "r-98371" > /dev/SStore0
# head -n 1 /dev/SStore0
   The answer is 42.
$ tail -n 1 /var/log/messages
...kernel: Source: 18446744072644022285
```

In other words, `r-98371` landed 13 bytes after the target address (landing before a target string is not as helpful, as reading terminates on a null byte). To protect against this case, extra spaces were added to the beginning of the flag string.

For a real-world use, one can simply plug-in random numbers until something interesting is found. For example:

```
# echo "r8" > /dev/SStore0; head -n 0 /dev/SStore0 | hexdump
```

Prints the following (hexdump used because the string is not printable):

```
0000000 c096 ffff ffff d528 c096 ffff ffff 000a
```

Of course, whether there is any useful data to be stolen, and where it might be, are both highly variable.

Micro, Homogeneous

The homogeneous results are more interesting. Simply unloading `monolith.ko` and loading `micro.ko` is not enough. A user-space server must also be running, or else any operation hangs. After running `micro` (the server written in C), the program behaves as expected, except that the following is printed to `stderr` instead of the kernel log:

```
# ./micro /dev/SSS0 &
# echo "r0" > /dev/SStore0; head -n 1 /dev/SStore0
String address:  140736312557712
```

Calculating an offset as follows, it is not possible to capture the flag.

```
$ echo $(( (140736312557712 - 18446744072644022272) / 17))
8278669299238
```

Attempting to use this offset will simply cause a segfault.

```
# echo "r8278669299238" > /dev/SStore0; cat /dev/SStore0
```

In this particular case, even attempting to go out of bounds by about 8 kilobytes triggered a segfault.

```
# echo "r500" > /dev/SStore0; cat /dev/SStore0
String address:  140726348461236^C[1]+ Segmentation fault ./micro
```

```
/dev/SSS0
```

However, even with memory protections in place, the C server is still vulnerable to a buffer overrun. Some data could still be exfiltrated. For example:

```
# echo "r-4" > /dev/SStore0
# head -n 1 /dev/SStore0
                               V
```

While the significance of this string (31 spaces and a 'V') isn't exactly clear, it does prove the feasibility of arbitrary data exfiltration. In a larger program, there may be a larger attack surface.

Micro, Heterogeneous

After killing the C service and starting the Guile service, it was not possible to exfiltrate any information. Simply showing situations in which data is not exfiltrated does not constitute proof. Instead, consider that the testing machine has only 8 GB of RAM. If it were possible to read directly from memory, attempting to read from memory which doesn't exist should cause an error. 8 GB = 8,589,934,592.

```
# echo "r9000000000" > /dev/SStore0; head -n 1 /dev/SStore0


# echo "i9000000000" > /dev/SStore0; echo "Out of memory?" > /dev/SStore0
# echo "r9000000000" > /dev/SStore0; head -n 1 /dev/SStore0
```

```
> Out of memory?
```

If it were possible to direct the program to read from or write to arbitrary memory locations, doing so on a memory address which is outside the bounds of the host machine's physical memory would surely fail. Clearly, the Guile micro-server doesn't suffer from the same weaknesses as the C micro-server and kernel module.

Thanks to Guile's safe implementation of both strings and (associative) arrays, memory safety is no longer of even the slightest concern.

Chapter 6

Conclusion

When the functionality of a program is not of critical security or performance concern, running it as ring zero presents clear dangers to the security and stability of the whole system.

The kernel module was able to read and write to arbitrary locations in memory, and was demonstrated to be able to exfiltrate string data easily.

Moving the vulnerable logic from kernel-space to user-space, as in the C server, was able to protect kernel memory, but it was still possible for a process to compromise its own memory.

The best solution was the hybrid approach. The Guile server was completely protected against out-of-bounds errors, as well as the runtime instability that might come from a segmentation fault. Although limited by the kernel-module, the Guile server also has the advantage of vastly easier and safer string manipulation.

The heterogeneous hybrid approach allows for the optimal compromise between performance and security. Long, convoluted, string or array-heavy code can be written in high-level languages and executed in user-space, while performance-critical code can be written in C and executed in user-space.

Bibliography

[1] "Security vulnerabilities (memory corruption) (cvss score ¿= 9)," CVE Details. (), [Online]. Available: https://www.cvedetails.com/vulnerability-list.php?vendor_id=0&product_id=0&version_id=0&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=1&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=9&cvssscoremax=0&year=0&month=0&cweid=0&order=1&trc=6982&sha=5829c45b747ab5143004640f312c7f72e5b102db (visited on 04/20/2023).

[2] B. Seri and G. Vishnepolsky, "The dangers of bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks.," Armis Labs, California, USA, Tech. Rep., 2017.

[3] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *CCS '15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS, Oct. 2015, pp. 414–425. DOI: https://doi.org/10.1145/2810103.2813637.

[4] C. Cutler, M. F. Kaashoek, and R. T. Morris, “The benefits and costs of writing a posix kernel in a high-level language,” in *OSDI ’18 Proceedings of the 13th USENIX conference on OS Design and Implementation*, Carlsbad, CA, USA: USENIX, 2018, pp. 89–105.