CERT/CC Blog

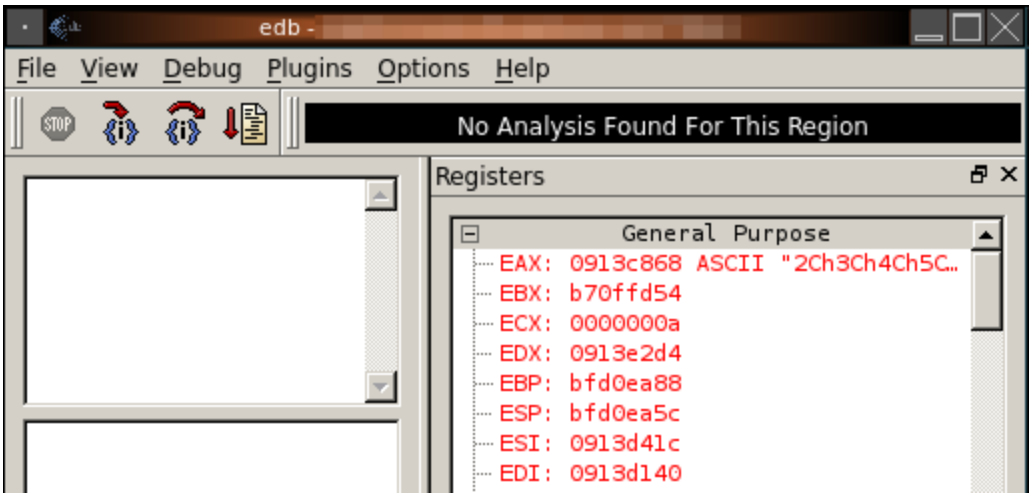
Vulnerability Insights

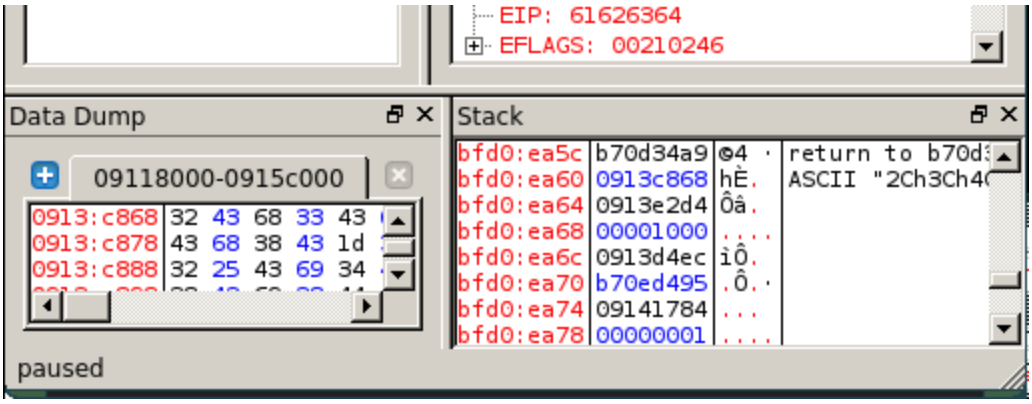
Feeling Insecure? Blame Your Parent!

POSTED ON FEBRUARY 3, 2014 BY WILL DORMANN [/AUTHOR/WILL-DORMANN] IN VULNERABILITY ANALYSIS [HTTPS://INSIGHTS.SEI.CMU.EDU/CERT/VULNERABILITY-ANALYSIS/]

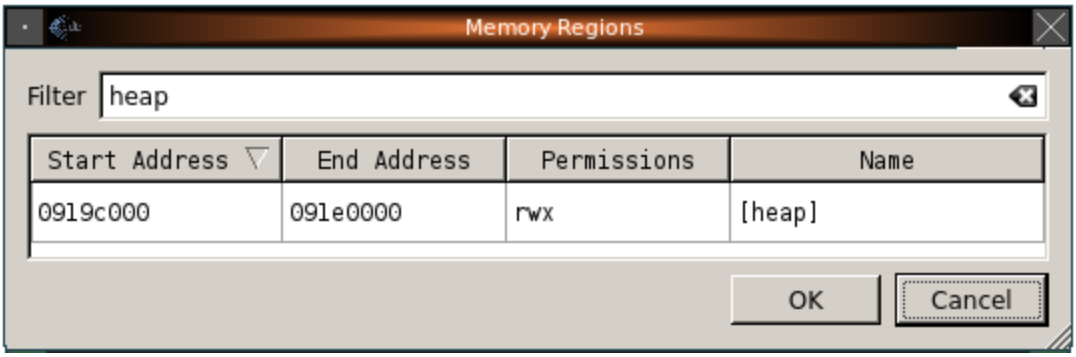
Hey, it's Will. I was recently working on a proof of concept (PoC) exploit using nothing but the CERT BFF [http://www.cert.org/vuls/discovery/bff.html] on Linux. Most of my experience with writing a PoC has been on Windows, so I figured it would be wise to expand to different platforms. However, once I got to the point of controlling the instruction pointer, I was surprised to discover that there was really nothing standing in the way of achieving code execution.

Without requiring additional work beyond doing a BFF minimization to Metasploit string [https://www.cert.org/blogs/certcc/2013/09/one_weird_trick_for_finding_mo.html], it was easy to tell what bytes were under my control and how I could get there:





Home run! I had control of `EIP` (`0x61626364` = "abcd"), I had a Metasploit string pattern, and also have a register that pointed there (`EAX`). Let's look at the memory protections for that location:



Well that can't be right, can it? The heap has Read, Write, and Execute permissions. It is a dangerous condition when any memory location is both writable and executable. Why is that? Using the case above as an example, this condition allows an attacker to simply jump to and execute bytes that are considered to be just data. NX [http://en.wikipedia.org/wiki/NX_bit#Linux] will not come into play if a memory region is marked as executable. (See also OpenBSD's W^X [http://en.wikipedia.org/wiki/W%5EX] policy.)

It took a little more digging to find out why the heap was executable. The platform that I'm using is UbuFuzz, which is Ubuntu 12.04.01 using an x86 Linux 3.2.0 kernel. On this platform if an executable stack is specified, such as by compiling the application with the gcc `-z execstack` option, the `READ_IMPLIES_EXEC` [http://lxr.free-electrons.com/source/fs/binfmt_elf.c?v=3.2&a=x86#L718] personality is set for the process. As is perhaps obvious by the name, memory locations that are readable are by default executable when this personality flag is set, including the heap.

The crashing process in question does **not** specify an executable stack in its ELF header, though:

```
GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4
```

If this process does not specify that it will have an executable stack, how did it end up with the `READ_IMPLIES_EXEC` personality? It doesn't explicitly use the personality() [http://man7.org/linux/man-pages/man2/personality.2.html] function either. Here's where things get a little

strange. The process that crashes is not invoked directly from a terminal, but rather it is spawned from a different application. Let's look at the stack properties specified by the parent executable:

```
GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
```

Well there's our culprit! As it turns out, on the x86 Linux platform, the `READ_IMPLIES_EXEC` personality bit is passed from a parent process to the child. This behavior is architecture-specific. A child process does not inherit `READ_IMPLIES_EXEC` on the amd64 architecture, for example. The `ADDR_NO_RANDOMIZE` flag, which is used to disable ASLR, is inherited by child processes on all architectures.

Conclusion

On a vanilla Linux platform, it may be difficult to statically determine which exploit mitigations may be enabled for any given process. With Linux on the x86 platform in particular, any given process may run without NX protections simply due to the properties of its parent process.

Thanks to The Pax Team [https://pax.grsecurity.net/] and Sebastian Krahmer of SUSE [https://www.suse.com/] for helping me with this topic!

About the Author

Will Dormann

Contact Will Dormann [https://www.sei.cmu.edu/contact.cfm]
Visit the SEI Digital Library for other publications by Will [https://resources.sei.cmu.edu/library/author.cfm?authorID=2547]
View other blog posts by Will Dormann [/author/will-dormann]