From Vulnerability to Exploit

Whether you use static analysis, dynamic analysis, or some combination of both to discover a problem with a piece of software, locating a potential problem or causing a program to melt down in the face of a fuzzer onslaught is just the first step. With static analysis in particular, you face the task of determining exactly how to reach the vulnerable code while the program is executing. Additional analysis followed by testing against a running program is the only way to confirm that your static analysis is correct. Should you provoke a crash using a fuzzer, you are still faced with the task of dissecting the fuzzer input that caused the crash and understanding any crash dumps yielded by the program you are analyzing. The fuzzer data needs to be dissected into the portions required strictly for code path traversal, and the portions that actually generate an error condition with the program. Knowing that you can crash a program is a far cry from understanding exactly why the program crashes. If you hope to provide any useful information to assist in patching the software, it is important to gain as detailed an understanding as possible about the nature of the problem. It would be nice to avoid this conversation:

Researcher: “Hey, your software crashes when I do this…”

Vendor: “Then don’t do that!

In favor of this one: Researcher: “Hey, you fail to validate the widget field in your octafloogaron application, which results in a buffer overflow in function umptiphratz. We’ve got packet captures, crash dumps, and proof of concept exploit code to help you understand the exact nature of the problem.”

Vendor: “All right, thanks, we will take care of that ASAP.”

Whether a vendor actually responds in such a positive manner is another matter. In fact, if there is one truth in the vulnerability research business, it's that dealing with vendors can be one of the least rewarding phases of the entire process. The point is that you have made it significantly easier for the vendor to reproduce and locate the problem and increased the likelihood that it will get fixed.

Exploitability

Crashability and exploitability are vastly different things. The ability to crash an application is, at a minimum, a form of denial of service. Unfortunately, depending on the robustness of the application, the only person whose service you may be denying could be you. For true exploitability, you are really interested in injecting and executing your own code within the vulnerable process. In the next few sections, we discuss some of the things to look for to help you determine whether a crash can be turned into an exploit.

Debugging for Exploitation

Developing and testing a successful exploit can take time and patience. A good debugger can be your best friend when trying to interpret the results of a program crash. More specifically, a debugger will give you the clearest picture of how your inputs have conspired to crash an application. Whether an attached debugger captures the state of a program when an exception occurs or you have a core dump file that can be examined, a debugger will give you the most comprehensive view of the state of the application when the problem occurred. For this reason, it is extremely important to understand what a debugger is capable of telling you and how to interpret that information.

Initial Analysis

Why did the program crash? Where did the program crash? These are the first two questions that need to be answered. The "why" you seek here is not the root cause of the crash, such as the fact that there is a buffer overflow problem in function xyz.

Instead, initially you need to know whether the program segfaulted or perhaps executed an illegal instruction. A good debugger will provide this information the moment the program crashes. A segfault might be reported by gdb as follows:

Program received signal SIGSEGV, Segmentation fault.

0x08048327 in main ()

Always make note of whether the address resembles user input in any way. It is common to use large strings of A's when attacking a program. One of the benefits to this is that the address 0x41414141 is easily recognized as originating from your input rather than correct program operation. Using the addresses reported in any error messages as clues, you next examine the CPU registers to correlate the problem to specific program activity.

General Register Analysis

If you haven't managed to take control of eip, the next step is to determine what damage you can do using other available registers. Disassembly of the program in the vicinity of eip should reveal the operation that caused the program crash. The ideal condition that you can take advantage of is a write operation to a location of your choosing. If the program has crashed while attempting to write to memory, you need to determine exactly how the destination address is being calculated. Each general-purpose register should be studied to see if it (a) contributes to the destination address computation and (b) contains user-supplied data. If both of these conditions hold, it should be possible to write to any memory location. The second thing to learn is exactly what is being written and whether you can control that value; if you can, you have the capability to write any value anywhere. Some creativity is required to utilize this seemingly minor capability to take control of the vulnerable program. The goal is to write your carefully chosen value to an address

that will ultimately result in control being passed to your shellcode. Common overwrite locations include saved return addresses, jump table pointers, import table pointers, and function pointers. Format string vulnerabilities and heap overflows both work in this manner because the attackers gain the ability to write a data value of their choosing (usually 4 bytes, but sometimes as little as 1 or as many as 8) to a location or locations of their choosing.

Improving Exploit Reliability

Another reason to spend some time understanding register content is to determine whether any registers point directly at your shellcode at the time you take control of eip. Since the big question to be answered when constructing an exploit is “What is the address of my shellcode?”, finding that address in a register can be a big help. As discussed in previous chapters, injecting the exact address of your shellcode into eip can lead to unreliable results since your shellcode may move around in memory. When the address of your shellcode appears in a CPU register, you gain the opportunity to do an indirect jump to your shellcode. Using a stack-based buffer overflow as an example, you know that a buffer has been overwritten to control a saved return address. Once the return address has been popped off the stack, the stack pointer continues to point to memory that was involved in the overflow and that could easily contain your shellcode.

The classic technique for return address specification is to overwrite the saved eip with an address that will point to your shellcode so that the return statement jumps directly into your code. While the return addresses can be difficult to predict, you do know that esp points to memory that contains your malicious input, because following the return from the vulnerable function, it points 4 bytes beyond the overwritten return address. A better technique for gaining reliable control would be to execute a jmp esp or call esp instruction at this point. Reaching your shellcode becomes a two-step process in this case. The first step is to overwrite the saved return address with the address of a jmp esp or call esp instruction. When the

exploitable function returns, control transfers to the jmp esp, which immediately transfers control back to your shellcode. This sequence of events is detailed in Figure 26-3.

A jump to esp is an obvious choice for this type of operation, but any register that happens to point to your user-supplied input buffer (the one containing your shellcode) can be used. Whether the exploit is a stack-based overflow, a heap overflow, or a format string exploit, if you can find a register that is left pointing to your buffer, you can attempt to vector a jump through that register to your code. For example, if you recognize that the esi register points to your buffer when you take control of eip, then a jmp esi instruction would be a very helpful thing to find.