Practical Analysis of Buffer Overflow Vulnerabilities and Privilege Escalation using Gsubmit

Kyle Brogle BUILDS *David House*

Abstract

Buffer overflows are a commonly overlooked attack vector that allow third parties to gain unauthorized control over system resources. In this presentation we outline the methodology behind a buffer overflow attack on 'gsubmit', a popular assignment submission program used within the BU CS department. A vulnerability resulting from unsafe system calls allows us to execute arbitrary commands as root user on servers running vulnerable versions of gsubmit, resulting in a system-wide loss of security via privilege escalation.

We demonstrate a proof-of-concept for this attack, and describe specific measures taken to patch this security vulnerability on newer versions of gsubmit. We also introduce "Low-hanging Fruit", a program that examines binaries and automatically identifies their susceptibility to buffer overflow attacks.

What is a buffer overflow?

A buffer overflow occurs when data written to a buffer, due to insufficient bounds checking, corrupts data values in memory addresses adjacent to the allocated buffer. This may result in erratic program behavior, including memory access errors, incorrect results, program termination (a crash), or a breach of system security. Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates. They are thus the basis of many software vulnerabilities and can be maliciously exploited.

Stack buffer overflows are caused when a program writes more data to a buffer located on the stack than there was actually allocated for that buffer. If the affected program is running with special privileges then the bug is a potential security vulnerability. If the stack buffer is filled with data supplied from an untrusted user then that user can corrupt the stack in such a way as to inject executable code into the running program and take control of the process.

Profiling Gsubmit

Gsubmit is a homework submission program used within the Computer Science Department at Boston University. Gsubmit functions by copying students' homework submission files into a directory system that is only viewable by authorized professors and teaching assistants. Gsubmit runs on server csa2 and is by far the most popular digital homework submission protocol within the Computer Science Department, making it a vital piece of software infrastructure. The high visibility of gsubmit and the free availability of its source code online makes it a prime target to potential attackers wishing to gain control of BU CS servers, student files, and other digital resources. In this presentation we present a security audit of gsubmit in hopes that the overall security of the program can be improved.

Identifying a Vulnerable Buffer

The first step in performing a buffer overflow attack is identifying a vulnerable buffer. When a user submits a file in gsubmit that has already been submitted, a function is called which prompts (y/n) whether the user wishes to overwrite their previous file. The (y/n) user response is stored in the buffer char Answer[2]; input to Answer[2] is left unchecked, and so providing an input longer than the buffer allows us to overwrite whatever memory exists above the local variable Answer[2] on the stack (fig. 1).

Switching Execution

Now that we have a vulnerable buffer, we can switch execution to other points in the process address space marked as executable. We accomplish this by overwriting the return address at the frame pointer with input from our vulnerable buffer. Due to various forms of linux/gcc protection, we are limited in our execution switch to just two memory regions: the region containing the gsubmit program code, and the region containing shared code libraries, such as libc.

Switching the execution point to libc is preferable as libc contains fork(), execve(), system(), and many other functions that would be useful to an attacker. However, PLT/GLOT separation security [1] prevents this "return to libc" attack, and we are ultimately stuck with switching execution to other gusbmit functions already in the process address space.

Identifying Unsafe Calls

Since we are limited in our execution switching to only those functions existing in the stack, we must identify unsafe calls within the gsubmit code if we have any hope of being successful in our attack. Analysis reveals that system() is called directly within the gsubmit code; the system() function is significant in that it executes a shell command passed to it and returns the result. In gsubmit, the argument passed to system() is built from minimal user input, and checked to make sure it is sanitized of anything that may be potentially harmful. The idea of exploiting this unsafe system() call is a step in the right direction, but we must get rid of gsubmit's sanitization routines if we are to proceed further.

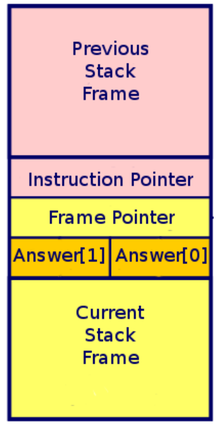
When disassembled, the instructions for gsubmit's system() call resemble:

```
lea     -0xbeef(%ebp),%eax
mov     %eax,(%esp)
call    0x8048dea <system@plt>
```

This tells us that the code will place the argument to the function system(), located at the address [frame pointer - `0xbeef`], into the register **`eax`**. **`eax`** will then be placed at the top of the stack, where a subsequent function call will expect its argument to be. The final instruction calls the system function with the specified argument.

Figure 1
An example Call Stack with gsubmit local variables (Answer) explicitly displayed.

Note that this stack grows down.



Hacking Around Sanitization

We can try to force gsubmit to call system() on a non-sanitized argument by modifying the argument directly on the stack. In fig. 1, notice that the saved frame pointer address can be overwritten, as well as the saved instruction pointer. If we can somehow insert the command we want to execute into the address space of gsubmit, we could modify the frame pointer in such a way that system() will be called with our command as the argument.

A new problem arises: how do we get our payload -- our illicit command to be called by system() -- into memory?
A simple way to insert this value into memory is to "submit" a file whose filename is the desired command. Since gsubmit uses this filename in many of the functions it calls, there are many places on the stack where the filename (our payload system command) is stored, making it easy for an attacker to access it once he or she has assumed control of the instruction and frame pointers.

The Exploit

Now that we have all the pieces of the puzzle...

(1) A file named as our payload argument (ls, whoami, etc) is submitted via gsubmit.
(2) The y/n buffer is overflowed, switching the execution point to lea -- the beginning of the system() call.
(3) The saved frame pointer is switched to point to the location where the payload argument resides (memory location of command + `0xbeef`).
(4) System() is called with the newly specified argument. All safety checks performed on the system argument are bypassed, since they came before the system call `lea` in memory.
(5) System() runs our payload command. Since the program is run with the SUID bit set, it runs with the privilege level of superuser (root).
(6) Our payload is executed with root permissions.

After the exploit is executed, a segmentation fault occurs since stack integrity is lost. This segmentation fault (and the resulting system log event) can be avoided by calling call_exit() after the system() function.

Low-Hanging Fruit

During the course of discovering, writing, and patching this exploit, we had the need for a quick way to determine possible vulnerabilities to buffer overflow attacks. While such tools exist, they fall into two main categories: ones that audit source code to find possible buffer overflows, and "active" tools that load a binary into memory and pepper the running program with random input in the hopes that it will cause a segmentation fault, signifying a problem (fuzzers). Since we wanted a general-purpose tool that could be used without having the source code of the program, and since debuggers can only be attached to a SUID program as root, we chose to write a tool to passively analyze the binary.

We authored a script entitled "Low-Hanging Fruit" which is used to identify buffer overflows in ELF binaries by checking for well-known vulnerable library routines. Low-Hanging Fruit is being written in C using the Unix Binary File Descriptor library in order to expand its uses as a tool for security-minded programmers and power users to quickly audit a binary for potential attack vectors.