Secure Programming Lecture 4: Memory Corruption II (Stack Overflows)

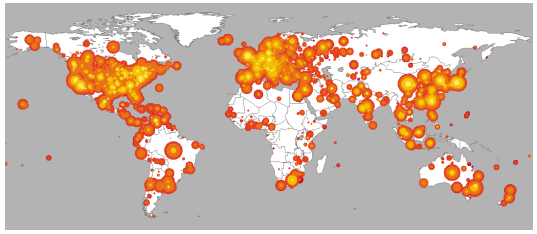
David Aspinall, Informatics @ Edinburgh

31st January 2017

Morris Worm (1998)

- Stack overflow attack on `fingerd`, long argument
- Infected 6000 machines, 10% of Internet in 1998.
- Impact: "Accidental" DoS. Estimated costs: $100k-$97m.

Code Red (2001)



- Stack overflow with crafted URL, overwriting exception handler
- Rapid spread. Infected 750,000 servers running MS IIS. Executed DoS attack on other websites. Estimated cost: $2bn.

Memory corruption

Buffer overflow remains a common vulnerability being discovered and exploited in commodity software.

- Simple cause:
  - putting *m* bytes into a buffer of size *n*, for *m>n*
  - corrupts the surrounding memory
- Simple fix:
  - check size of data before/when writing

Buffer overflow *exploits*, where the memory corruption is tailored to perform something specific attacker wants to do, can be technically very complex.

We'll study simple examples to explain how devastating they can be, starting with **classic stack overflows**.

Examples will use Linux/x86 to demonstrate; principles are similar on other OSes/architectures.

Is stack overflow still a problem?

Can we eradicate vulnerability types?

Is stack overflow still a problem?

Search Results (Refine Search)

There are **24** matching records.

Displaying matches **1** through **20**.

Search Parameters:

- **Keyword (text search):** stack overflow
- **Search Type:** Search Last 3 Months
- **Contains Software Flaws (CVE)**

CVE-2016-8807

Summary: For the NVIDIA Quadro, NVS, and GeForce products, NVIDIA Windows GPU Display Driver R340 before 342.00 and R375 before 375.63 contains a vulnerability in the kernel mode layer (nvlddmkm.sys) handler for DxgDdiEscape ID 0x10000e9 where a value is passed from an user to the driver is used without validation as the size input to memcpy() causing a stack buffer overflow, leading to denial of service or potential escalation of privileges.

Published: 11/8/2016 3:59:20 PM

CVSS Severity: v3 - 7.8 HIGH v2 - 7.2 HIGH

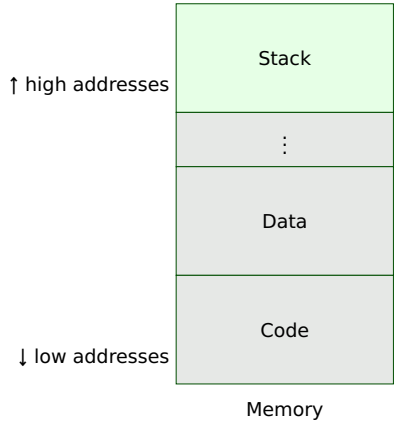
CVE-2016-9052

Summary: An exploitable stack-based buffer overflow vulnerability exists in the querying functionality of Aerospike Database Server 3.10.0.3. A specially crafted packet can cause a stack-based buffer overflow in the function as_sindex__simatch_by_iname resulting in remote code execution. An attacker can simply connect to the port to trigger this vulnerability.

Published: 1/26/2017 4:59:00 PM

CVSS Severity: v3 - 9.8 CRITICAL v2 - 7.5 HIGH

How the stack works (simplified)



Corrupting stack variables

Local variables are put close together on the stack.

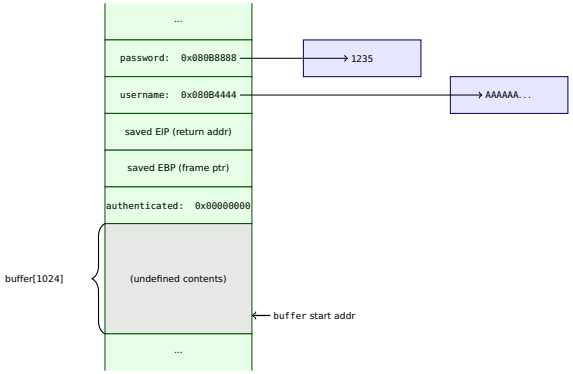
- If a stray write goes beyond the size of one variable
- . . . it can corrupt another

Application scenario

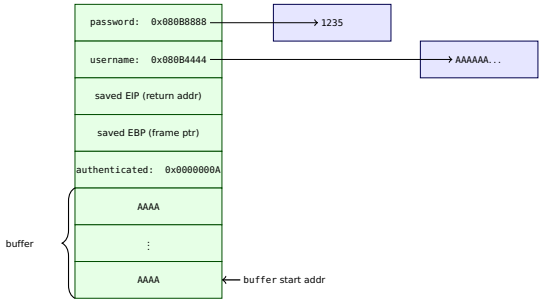
```c
int authenticate(char *username, char *password) {

  int authenticated; // flag, non-zero if authenticated
  char buffer[1024]; // buffer for username

  authenticated = verify_password(username, password);

  if (authenticated == 0) {
    sprintf(buffer,
        "Incorrect password for user %s\n",
        username);
    log("%s",buffer);
  }
  return authenticated;
}
```

- Vulnerability in `authenticate()` call to `sprintf()`.
- If the `username` is longer than 1023 bytes, data will be written past the end of the `buffer`.

Possible stack frame before exploit



Stack frame after exploit



- If `username` is >1023 letters long, `authenticated` is corrupted and may be set to non-zero.
- E.g., char 1024='\n', the low byte becomes 10.

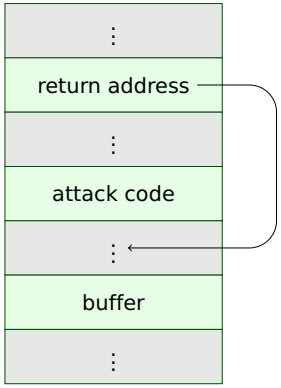
Local variable corruption remarks

Tricky in practice:

- location of variables may not be known
  - memory addresses can vary between invocations
  - C standards don't specify stack layout
  - compiler moves things around, optimises layout
- effect depends on behaviour of application code

A more predictable, *general* attack works by corrupting the fixed information in every stack frame: the frame pointer and return address.

Stack overflow exploit



The malicious argument overwrites all of the space allocated for the buffer, all the way to the return address location.
The return address is altered to point back into the stack, somewhere before the attack code.
Typically, the attack code executes a shell.

Attacker controlled execution

By over-writing the return address, the attacker may either:

1. set it to point to some known piece of the application code, or code inside a shared library, which achieves something useful, or
2. supply his/her own code somewhere in memory, which may do anything, and arrange to call that.

The second option is the most general and powerful.

How does it work?

Arbitrary code exploit

The attacker takes these steps:

1. store executable code somewhere in memory
2. use stack overflow to direct execution there
3. code does something useful to attacker

The attack code is known as **shellcode**. Typically, it launches a shell or network connection.

Shellcode is ideally:

- small and self-contained
- position independent
- free of ASCII NUL (0x00) characters

Question. Why?

Arbitrary code exploit

1. store executable code somewhere in memory
2. use stack overflow to re-direct execution there
3. **code does something useful to attacker**

Building shellcode

Consider spawning a shell in Unix. The code looks like this:

```
#include <unistd.h>
...
char *args[] = { "/bin/sh", NULL };
execve("bin/sh", args, NULL)
```

- `execve()` is part of the Standard C Library, `libc`
- it starts a process with the given name and argument list and the environment as the third parameter.

We want to write (relocatable) assembly code which does the same thing: constructing the argument lists and then invoking the execve function.

Invoking system calls

To execute a *library* function, the code would need to find the location of the function.

- for a dynamically loaded library, this requires ensuring it is loaded into memory, negotiating with the linker
- this would need quite a bit of assembly code

It is easier to make a *system call* directly to the operating system.

- luckily, `execve()` is a library call which corresponds exactly to a system call.

Invoking system calls

Linux system calls (32 bit x86) operate like this:

- Store parameters in registers EBX, ECX, . . .
- Put the desired system call number into AL
- Use the interrupt `int 128` to trigger the call

Invoking a shell

Here is the assembly code for a simple system call invoking a shell:

```
        .section .rodata    # data section
args:
        .long arg           # char *["/bin/sh"]
        .long 0             #
arg:
        .string "/bin/sh"

        .text
        .globl  main
main:
        movl    $arg, %ebx
        movl    $args, %ecx
        movl    $0, %edx
        movl    $0xb, %eax
        int     $0x80       # execve("/bin/sh",["/bin/sh"],NULL)
        ret
```

From assembly to shellcode

However, this is not yet quite shellcode: it contains hard-wired (absolute) addresses and a data section.

Question. How could you turn this into position independent code without separate data?

From assembly to shellcode

Moreover, we need to find the binary representation of the instructions (i.e., the compiled code).

This will be the *data* that we can then feed back into our attack.

```
$ gcc shellcode.s -o shellcode.out
$ objdump -d shellcode.out
...
080483ed <main>:
 80483ed:   bb a8 84 04 08          mov    $0x80484a8,%ebx
 80483f2:   b9 a0 84 04 08          mov    $0x80484a0,%ecx
 80483f7:   ba 00 00 00 00          mov    $0x0,%edx
 80483fc:   b8 0b 00 00 00          mov    $0xb,%eax
 8048401:   cd 80                   int    $0x80
 8048403:   c3                      ret
```

- We take the hex op code sequence `bb a8 84...` etc and encode it as a string (or URL, filename, etc) to feed into the program as malicious input.

Arbitrary code exploit

1. **store executable code somewhere in memory**
2. **use stack overflow to direct execution there**
3. code does something useful to attacker

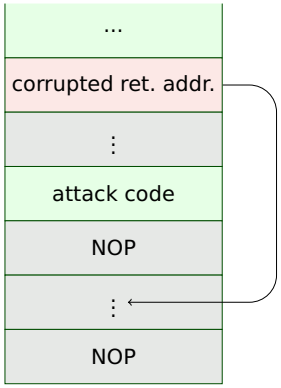
Two options:

- shellcode on stack
- shellcode in another part of the program data

Problem in both cases is :

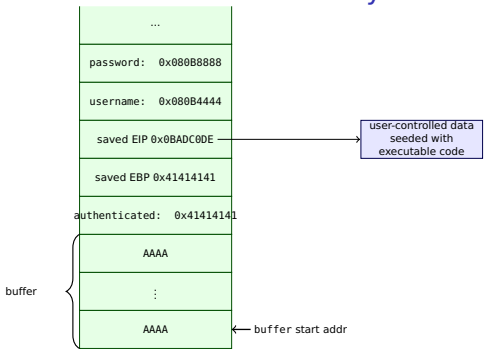
- how to find out where the code is?

Attack code on stack: the NOP sled



The exact address of the attack code in the stack is hard to guess.
The attacker can increase the chance of success by allowing a range of addresses to work.
The overflow uses a *NOP sled*, which the CPU execution "lands on", before being directed to the attack code.

Attack code elsewhere in memory



- Various (intricate) possibilities
- E.g., modifying function pointers or corrupting caller's saved frame pointer

Putting it together

The best way to understand this attack is to try it out!

We will carry out these steps in **Lab Session 1**.

Review questions

Stack overflows

- Explain how uncontrolled memory writing can let an attacker corrupt the value of local variables.
- Explain how an attacker can exploit a stack overflow to execute arbitrary code.
- Draw an example stack during a stack overflow attack with a NOP sled, showing some possible addresses for the shellcode location and return address.

Coming next

We'll continue looking at some other kinds of overflow attacks, then consider some general protection mechanisms.

References and credits

This lecture included examples from:

- M. Dowd, J. McDonald and J. Schuh. *The Art of Software Security Assessment*, Addison-Wesley 2007. The local variable example code and stack pictures are adatpted from Chapter 5.

- The Computer History Museum picture is from Intel Free Press.
- The picture of the Code Red spread is from the CAIDA Code Red analysis.