p2

Jeff Chase
Duke University
Smashing The Stack For Fun And Profit

Aleph One

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This
main() {
    int *ret;
    ret=(int *)&ret +2;
    printf("Shellcode length=%d\n",strlen(shellcode));
    (*ret) = (int)shellcode;
}
void main(int argc, char *argv[]) {
    char buffer[512];

    if (argc > 1)
        strcpy(buffer, argv[1]);
}
VAS example (32-bit)

- The program uses virtual memory through its process’ Virtual Address Space:
  - An addressable array of bytes…
  - Containing every instruction the process thread can execute…
  - And every piece of data those instructions can read/write…
    - i.e., read/write == load/store on memory
- Partitioned into logical segments (regions) with distinct purpose and use.
- Every memory reference by a thread is interpreted in the context of its VAS.
  - Resolves to a location in machine memory
Memory segments: a view from C

- **Globals:**
  - Fixed-size segment
  - Writable by user program
  - May have initial values

- **Text (instructions)**
  - Fixed-size segment
  - Executable
  - Not writable

- **Heap and stack**
  - Variable-size segments
  - Writable
  - Zero-filled on demand
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)
- Random stack offset
- RLIMIT_STACK (e.g., 8MB)
- Random mmap offset

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap
- Program break
- brk
- start_brk
- Random brk offset

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

0xc0000000 == TASK_SIZE

0x08048000
0
Smashing the Stack for Fun and Profit
A stack frame (x86)

http://duartes.org/gustavo/blog/post/journey-to-the-stack/
By looking at the assembly language output we see that the call to function() is translated to:

```
pushl $3
pushl $2
pushl $1
call function
```

This pushes the 3 arguments to function backwards into the stack, and calls function(). The instruction 'call' will push the instruction pointer (IP) onto the stack. We'll call the saved IP the return address (RET). The first thing done in function is the procedure prolog:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

This pushes EBP, the frame pointer, onto the stack. It then copies the current SP onto EBP, making it the new FP pointer. We'll call the saved FP pointer SFP. It then allocates space for the local variables by subtracting their size from SP.

Smashing the Stack for Fun and Profit
CPU Registers

- esp: stack pointer
- ebp: frame pointer
- eax: return value

Stack:
- local_buffer: f o o b a r 0 0 ?
- local1: 0 0 0 0 0 0
- saved ebp
- return address: 0 0 0 2 0 0
- param1: 512 0 0 0 0 0
- param2: 65536 0 0 0 0 1 0 0

Stack growth: left to right
Higher memory addresses: right to left

http://duartes.org/gustavo/blog/post/journey-to-the-stack/
Smashing the return address

```c
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    int *ret;

    ret = buffer1 + 12;
    (*ret) += 8;
}

void main() {
    int x;

    x = 0;
    function(1,2,3);
    x = 1;
    printf("%d\n",x);
}
```

Smashing the Stack for Fun and Profit
Where is that stack?

```c
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
void main() {
    printf("0x%lx\n", get_sp());
}
```

```
[aleph1]$ ./sp
0x8000470
[aleph1]$ 
```

Smashing the Stack for Fun and Profit
Exploit development can lead to serious headaches if you don't adequately account for factors that introduce *non-determinism* into the debugging process. In particular, the stack addresses in the debugger may not match the addresses during normal execution. This artifact occurs because the operating system loader places both environment variables and program arguments *before* the beginning of the stack:

Since your vulnerable program does not take any arguments, the environment variables are likely the culprit. Make sure they are the same in both invocations, in the shell and in the debugger.

P2: break a simple web server

- The web server is based on:
  - */c-samples/buggyserver.c

- This server has a bug that makes it vulnerable to a stack smash attack (previously discussed).

- Stack smash attacks may enable remote execution of code chosen by the attacker, to “own” the web server.

- Each group gets their own instance to attack. If you crack it you get the points.

- Test your talents, but please do not abuse them.

- These attacks have unleashed untold pain into the world…and it never stops.
CVE-2015-7547: glibc getaddrinfo stack-based buffer overflow

Posted: Tuesday, February 16, 2016

Posted by Fermin J. Serna, Staff Security Engineer and Kevin Stadmeyer, Technical Program Manager

Have you ever been deep in the mines of debugging and suddenly realized that you were staring at something far more interesting than you were expecting? You are not alone! Recently a Google engineer noticed that their SSH client segfaulted every time they tried to connect to a specific host. That engineer filed a ticket to investigate the behavior and after an intense investigation we discovered the issue lay in glibc and not in SSH as we were expecting.

Thanks to this engineer’s keen observation, we were able determine that the issue could result in remote code execution. We immediately began an in-depth analysis of the issue to determine whether it could be exploited, and possible fixes. We saw this as a challenge, and after some intense hacking sessions, we were able to craft a full working exploit!
Stack smash defenses

- Modern systems have various defenses.
  - **NX: no-execute segments.** The classic attack injects code onto a buffer that resides on the stack, and overwrites a return address to branch to the injected code. We can make this harder by disabling execute privilege on the stack segment.
  - **ASLR: address space layout randomization.** The attacker guesses where the stack resides in order to overwrite a frame’s return address to branch to injected code. Randomizing the layout makes this harder.

- These have been disabled in the web server instances.
Server listens on a socket

```c
struct sockaddr_in socket_addr;
sock = socket(PF_INET, SOCK_STREAM, 0);
    int on = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);

memset(&socket_addr, 0, sizeof socket_addr);
socket_addr.sin_family = PF_INET;
socket_addr.sin_port = htons(port);
socket_addr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&socket_addr, sizeof socket_addr) < 0) {
    perror("couldn't bind");
    exit(1);
}
listen(sock, 10);
```
Accept loop: trivial example

while (1) {
    int acceptsock = accept(sock, NULL, NULL);
    char *input = (char *)malloc(1024*sizeof (char));
    recv(acceptsock, input, 1024, 0);
    int is_html = 0;
    char *contents = handle(input,&is_html);
    free(input);

    …send response…

    close(acceptsock);
}

If a server is listening on only one port/socket (“listener”), then it can skip the select/poll/epoll.
Illustration only
Send HTTP/HTML response

const char *resp_ok = "HTTP/1.1 200 OK
Server: BuggyServer/1.0\n";
const char *content_html = "Content-type: text/html\n\n";

send(acceptsock, resp_ok, strlen(resp_ok), 0);
send(acceptsock, content_html, strlen(content_html), 0);
send(acceptsock, contents, strlen(contents), 0);
send(acceptsock, "\n", 1, 0);

free(contents);