CS165 – Computer Security

Midterm Review

November 3, 2025

Midterm Structure

- Three sections
  - 7 multiple choice – answer 5 (6pts each)
    - Choose the best answer from a list
  - 4 essay-ish – answer 3 (10pts each)
    - Answer 1 or 2 conceptual questions
    - Free form – 2-3 sentences
  - 3 “constructions” (40pts)
    - Scenarios with problem solving
    - 3-4 sub-questions
- Watch the time – answer the questions you know first

Midterm Scope

- Up to and including the "Defenses lecture"
  - Vulnerabilities
  - History (Attack surface)
  - Memory errors (spatial, type, temporal)
  - Buffer overflow attacks (stack, P1)
  - Basic Defenses (for code injection, e.g., canaries)
  - Heap attacks (P1)
  - ROP
  - ROP defenses (safe stack and CFI)

Homework

- 1. What is necessary for a software flaw (e.g., memory error)?
  - a) The flaw must be accessible to an adversary.
  - b) An adversary must be able to exploit the flaw.
  - c) Both a) and b)
- 2. Which of the following describes an attack on availability?
  - a) It is hard to notice.
  - b) It can stop legitimate users from using a service
  - c) It can only happen due to a network denial-of-service attack.

Homework

- 1. What is necessary for a software flaw (e.g., memory error)?
  - a) The flaw must be accessible to an adversary.
  - b) An adversary must be able to exploit the flaw.
  - c) Both a) and b)
- 2. Which of the following describes an attack on availability?
  - a) It is hard to notice.
  - b) It can stop legitimate users from using a service
  - c) It can only happen due to a network denial-of-service attack.

Homework (Part 1)

- 3. Why is computer security about looking at corner cases of a program?
  - a) Because vulnerabilities are triggered by inputs that are commonly observed in typical workloads.
  - b) Because security problems cannot occur in common cases of a program.
  - c) Because many security vulnerabilities are hidden and hard to discover.
- 4. Which statement best describes a spatial error like a buffer overflow? (read carefully, not necessarily an error)
  - a) A referent (i.e., pointer) assigned to an allocated region may be used to read outside that allocated region.
  - b) Allows a memory write to happen outside one allocated memory region.
  - c) A pointer is used in a memory operation before being assigned to an allocated region.

Homework (Part 1)

- 3. Why is computer security about looking at corner cases of a program?
  - a) Because vulnerabilities are triggered by inputs that are commonly observed in typical workloads.
  - b) Because security problems cannot occur in common cases of a program.
  - c) Because many security vulnerabilities are hidden and hard to discover.
- 4. Which statement best describes a spatial error like a buffer overflow? (read carefully, not necessarily an error)
  - a) A referent (i.e., pointer) assigned to an allocated region may be used to read outside that allocated region.
  - b) Allows a memory write to happen outside one allocated memory region.

  (lots of memory ops occur outside of a region)
  - c) A pointer is used in a memory operation before being assigned to an allocated region.

Homework (Part 1)

- 5. Which statement best describes a temporal error?
  - a) A memory region is read before a pointer is assigned to reference that region.
  - b) A pointer is assigned to an allocated region of another data type.
  - c) A pointer is used in a memory operation before being assigned to an allocated region.
- 6. What happens when we cast on object of type A to an object of type B in the C programming language?
  - a) Assign a pointer to the object that interprets the object’s memory layout according to type B.
  - b) Reformat the memory layout of the object (originally of type A) to the format of type B.
  - c) Casts between different types are not allowed in the C programming language.

Homework (Part 1)

- 5. 5. Which statement best describes a temporal error?
  - a) A memory region is read before a pointer is assigned to reference that region. (may be caused by a spatial error)
  - b) A pointer is assigned to an allocated region of another data type.
  - c) A pointer is used in a memory operation before being assigned to an allocated region.
- 6. What happens when we cast on object of type A to an object of type B in the C programming language?
  - a) Assign a pointer to the object that interprets the object's memory layout according to type B.
  - b) Reformat the memory layout of the object (originally of type A) to the format of type B.
  - c) Casts between different types are not allowed in the C programming language.

Homework (Part 1)

- 7. What is a security flaw that may be caused because of the limitations of strncpy?
  - a) Place a null terminator at the beginning of the written memory.
  - b) Create a string that lacks a null-terminator.
  - c) Write outside the destination's memory region.
- 8. What must an adversary modify via a memory error permits to launch a control-flow hijack?
  - a) a function pointer
  - b) a data pointer
  - c) a function's code

Homework (Part 1)

- 7. What is a security flaw that may be caused because of the limitations of strncpy?
  - a) Place a null terminator at the beginning of the written memory.
  - b) Create a string that lacks a null-terminator.
  - c) Write outside the destination's memory region. (if N is too big)
- 8. What must an adversary modify via a memory error permits to launch a control-flow hijack?
  - a) a function pointer
  - b) a data pointer
  - c) a function's code

Homework (Part 1)

- 9. What is the advantage of applying the “%ms” format identifier in scanf?
  - a) Avoids the program running out of memory.
  - b) Automatically performs all allocations and deallocations for the string object.
  - c) Allocates a larger buffer when the input exceeds the memory allocated for the string.
- 10. Which instruction pushes a return address on the stack?
  - a) pop
  - b) call
  - c) ret

Homework (Part 1)

- 9. What is the advantage of applying the “%ms” format identifier in scanf?
  - a) Avoids the program running out of memory.
  - b) Automatically performs all allocations and deallocations for the string object. (does not perform deallocations)
  - c) Allocates a larger buffer when the input exceeds the memory allocated for the string.
- 10. Which instruction pushes a return address on the stack?
  - a) pop
  - b) call
  - c) ret

Homework (Part II)

- IV. Briefly describe the purpose the following instructions and what they do:
  - 1) call
  - 2) leave
  - 3) ret

Homework (Part II)

- IV. Briefly describe the purpose the following instructions and what they do:
  - 1) call
  - Push the return address (address of the next instruction to the call instruction) onto the top of the stack and jump to the target address to execute (%eip changed to the target address specified in call instruction)
  - 2) leave
  - 3) ret

Homework (Part II)

- IV. Briefly describe the purpose the following instructions and what they do:
  - 1) call
  - Push the return address (address of the next instruction to the call instruction) onto the top of the stack and jump to the target address to execute (%eip changed to the target address specified in call instruction)
  - 2) leave
  - Copies the frame pointer %ebp (register) to %esp (register), which releases the stack frame. The old frame pointer (at the top of the stack) is then popped (restored) into %ebp (register).
  - 3) ret

Homework (Part II)

- IV. Briefly describe the purpose the following instructions and what they do:
  - 1) call
  - Push the return address (address of the next instruction to the call instruction) onto the top of the stack and jump to the target address to execute (%eip changed to the target address specified in call instruction)
  - 2) leave
  - Copies the frame pointer %ebp (register) to %esp (register), which releases the stack frame. The old frame pointer (at the top of the stack) is then popped (restored) into %ebp (register).
  - 3) ret
  - Pop the stack (i.e., value referenced by the %esp, which should be the return address) and put it in %eip (so the program jumps to the return address and start executing)

Homework (Part III)

- Suppose pointer x is assigned to an object of type A.

```
x = (A *) malloc(sizeof(A));
```

- When the object is freed (free(x)), x is then used after an object of type B has been allocated in the location of the former object of type A (i.e., x now references the object of type B). Below are the type definitions for types A and B.

```
typedef struct a {int f1; char f2[8]; int f3; int f4; char f5[8];} A;
typedef struct b {char x3[3]; char *x1; int x3; char x4[12]; char *x5;} B;
```

- Assuming int's and pointers are 4 bytes, devise operations on x will update a pointer in the new object of type B. Be creative.

Homework (Part III)

- Suppose pointer x is assigned to an object of type A.

```
x = (A *) malloc(sizeof(A));
```

- When the object is freed (free(x)), x is then used after an object of type B has been allocated in the location of the former object of type A (i.e., x now references the object of type B). Below are the type definitions for types A and B.

```
typedef struct a {int f1; char f2[8]; int f3; int f4; char f5[8];} A;
typedef struct b {char x3[3]; char *x1; int x3; char x4[12]; char *x5;} B;
```

- Assuming int's and pointers are 4 bytes, devise operations on x will update a pointer in the new object of type B. Be creative.
  - Goal: update a pointer field (x1 or x5); from corresponding field in x

Homework (Part III)

- Suppose pointer x is assigned to an object of type A.

```
x = (A *) malloc(sizeof(A));
```

- When the object is freed (free(x)), x is then used after an object of type B has been allocated in the location of the former object of type A (i.e., x now references the object of type B). Below are the type definitions for types A and B.

```
typedef struct a {int f1; char f2[8]; int f3; int f4; char f5[8];} A;
typedef struct b {char x3[3]; char *x1; int x3; char x4[12]; char *x5;} B;
```

- Assuming int's and pointers are 4 bytes, devise operations on x will update a pointer in the new object of type B. Be creative.

<table>
<tr><td>f1 (4)</td><td colspan="2">f2 (8)</td><td>f3 (4)</td><td>f4 (4)</td><td colspan="2">f5 (8)</td></tr>
<tr><td>x3 (4)</td><td>x1 (4)</td><td>x3 (4)</td><td colspan="3">x4 (12)</td><td>x5 (4)</td></tr>
</table>

Homework (Part III)

- Suppose pointer x is assigned to an object of type A.

```
x = (A *) malloc(sizeof(A));
```

- When the object is freed (free(x)), x is then used after an object of type B has been allocated in the location of the former object of type A (i.e., x now references the object of type B). Below are the type definitions for types A and B.

```
typedef struct a {int f1; char f2[8]; int f3; int f4; char f5[8];} A;
typedef struct b {char x3[3]; char *x1; int x3; char x4[12]; char *x5;} B;
```

- Assuming int's and pointers are 4 bytes, devise operations on x will update a pointer in the new object of type B. Be creative.

<table>
<tr><td>f1 (4)</td><td>attack</td><td>(8)</td><td>f3 (4)</td><td>f4 (4)</td><td>f5</td><td>attack</td></tr>
<tr><td>x3 (4)</td><td>x1 (4)</td><td>x3 (4)</td><td colspan="3">x4 (12)</td><td>x5 (4)</td></tr>
</table>

Homework (Part IV)

- 1. 1.Suppose we want to modify the value of the variable x, which is directly above the buffer y on the stack (NOTE: not guaranteed to be that way, but please assume that here). What is the minimum number of bytes that have to be written in the statement on line 6 to modify x?

```
1: int main( int argc, char *argv[] )
2: {
3:   int x;
4:   char y[40];
5:
6:   strcpy(y+8, argv[1]);
7:   use(x);
8: }
```

Homework (Part IV)

- 1. 1.Suppose we want to modify the value of the variable x, which is directly above the buffer y on the stack (NOTE: not guaranteed to be that way, but please assume that here). What is the minimum number of bytes that have to be written in the statement on line 6 to modify x? 40+1-8 = 33 (size of y (40), less the offset in line 6 (8), plus one to impact x)

```
1: int main( int argc, char *argv[] )
2: {
3:   int x;
4:   char y[40];
5:
6:   strcpy(y+8, argv[1]);
7:   use(x);
8: }
```

Homework (Part IV)

- 2. Identify one line in the program below (victim) that reads adversary input (i.e., is its attack surface), when an adversary can modify files A and B.

```
1: int main( void )
2: {
3:   FILE *fp1, *fp2;
4:   char input1[10], input2[10];
5:
6:   fp1 = fopen(A, "w");   // open for writing only
7:   fp2 = fopen(B, "r");   // open for reading only
8:
9:   fread(input1, ..., fp1);  // this would be an error
10:  fread(input2, ..., fp2);  // read adversary input here
11:  use(input1, input2);      // use it here, but credited
12: }
```

Homework (Part IV)

- 2. Identify one line in the program below (victim) that reads adversary input (i.e., is its attack surface), when an adversary can modify files A and B.

```
1: int main( void )
2: {
3:   FILE *fp1, *fp2;
4:   char input1[10], input2[10];
5:
6:   fp1 = fopen(A, "w");   // open for writing only. (so cannot read input from fp1)
7:   fp2 = fopen(B, "r");   // open for reading only
8:
9:   fread(input1, ..., fp1);  // this would return an error
10:  fread(input2, ..., fp2);  // read adversary input here
11:  use(input1, input2);      // use adversary input here
12: }
```

Homework (Part V)

```
1: char *p;

2: p = (char *) malloc(size);

3: len = snprintf(p, size, ``%s'', adv input);

4: free(p);
```

- 1. Is there a spatial or temporal memory error in this code? Why or why not.
- 2. Suppose the statements on lines 3 and 4 are switched? Explain any problem that could be caused.
  - NOTE: Assume the program is multi-threaded.

Homework (Part V)

```
1: char *p;
2: p = (char *) malloc(size);
3: len = snprintf(p, size, ``%s'', adv input);
4: free(p);
```

- 1. Is there a spatial or temporal memory error in this code? No.

  Why or why not.
  - **Spatial**: snprintf restricts the write to the ‘size’ allocated and ensures a null-terminator is placed – no spatial error
- 2. Suppose the statements on lines 3 and 4 are switched?

  Explain any problem that could be caused.
  - NOTE: Assume the program is multi-threaded.

Homework (Part V)

```
1: char *p;
2: p = (char *) malloc(size);
3: len = snprintf(p, size, ``%s'', adv input);
4: free(p);
```

- 1. Is there a spatial or temporal memory error in this code? No. Why or why not.
  - Spatial: snprintf restricts the write to the ‘size’ allocated and ensures a null-terminator is placed – no spatial error
  - Temporal: no use before initialization of ‘p’. No use after free of ‘p’. No temporal error
- 2. Suppose the statements on lines 3 and 4 are switched? Explain any problem that could be caused.
  - NOTE: Assume the program is multi-threaded.

Homework (Part V)

```
1: char *p;
2: p = (char *) malloc(size);
3: len = snprintf(p, size, ``%s'', adv input);
4: free(p);
```

- 1. Is there a spatial or temporal memory error in this code?
  Why or why not.
  - Spatial: snprintf restricts the write to the ‘size’ allocated and ensures a null-terminator is placed – no spatial error
  - Temporal: no use before initialization of ‘p’. No use after free of ‘p’. No temporal error
  - Requirements of a legal C string operation would require checking for truncation
- 2. Suppose the statements on lines 3 and 4 are switched?
  Explain any problem that could be caused.
  - NOTE: Assume the program is multi-threaded.

Homework (Part V)

```
1: char *p;
2: p = (char *) malloc(size);
3: len = snprintf(p, size, ``%s'', adv input);
4: free(p);
```

- 1. Is there a spatial or temporal memory error in this code? Why or why not.
- 2. Suppose the statements on lines 3 and 4 are switched? Explain any problem that could be caused.
  - NOTE: Assume the program is multi-threaded.
  - Could perform the write to memory location ‘p’ after it is freed. Why is that a problem?

Homework (Part V)

```
1: char *p;

2: p = (char *) malloc(size);

3: len = snprintf(p, size, ``%s'', adv input);

4: free(p);
```

- 1. Is there a spatial or temporal memory error in this code? Why or why not.
- 2. Suppose the statements on lines 3 and 4 are switched? Explain any problem that could be caused.
  - NOTE: Assume the program is multi-threaded.
  - Could perform the write to memory location 'p' after it is freed. Why is that a problem?
  - Other thread could allocate memory at p between statements 4 and 3, causing p to be used to write to another object.

Homework (Part VI)

```
char *getline()
{
    char buf[8];
    char *result;
    scanf("%s", buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

```
08048524 <getline>:
8048524: 55 push %ebp
8048525: 89 e5 mov %esp,%ebp
8048527: 83 ec 10 sub $0x10,%esp
804852a: 56 push %ecx
804852b: 53 push %edi
Diagram stack at this point
804852c: 83 c4 f4 add $0xfffffff4,%esp
804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
8048532: 53 push %ebx
8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.

Homework (Part VI)

```
char *getline()
{
    char buf[8];
    char *result;
    scanf("%s", buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

```
1 08048524 <getline>:
2 8048524: 55 push %ebp
3 8048525: 89 e5 mov %esp,%ebp
4 8048527: 83 ec 10 sub $0x10,%esp
5 804852a: 56 push %ecx
6 804852b: 53 push %edi
Diagram stack at this point
7 804852c: 83 c4 f4 add $0xfffffff4,%esp
8 804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
9 8048532: 53 push %ebx
10 8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (1) Fill in the diagram below indicating as much as you can about the stack just after executing the instruction at line 6 in the disassembly.

Homework (Part VI)

```
+-------------+
|08 04 ab 8c  | Return Address
+-------------+
|bf ff fa 60  | Saved %ebp
+-------------+
| buf[4-7)    |
+-------------+
| buf[0-3)    |
+-------------+
| unused      | stack adjusts
+-------------+
|result [0-3] | by 0x10 bytes
+-------------+
| 00 00 00 08 | Saved %ecx
+-------------+
| 00 00 00 03 | Saved %edi,
%esp references this location
+-------------+
```

```
1 08048524 <getline>:
2 8048524: 55 push %ebp
3 8048525: 89 e5 mov %esp,%ebp
4 8048527: 83 ec 10 sub $0x10,%esp
5 804852a: 56 push %ecx
6 804852b: 53 push %edi
Diagram stack at this point
7 804852c: 83 c4 f4 add $0xfffffff4,%esp
8 804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
9 8048532: 53 push %ebx
10 8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (1) Fill in the diagram below indicating as much as you can about the stack just after executing the instruction at line 6 in the disassembly.

Homework (Part VI)

```
4 char *getline()
5 {
6     char buf[8];
7     char *result;
8     scanf("%s", buf);
9     result = malloc(strlen(buf));
10    strcpy(result, buf);
11    return result;
12 }
```

```
1 08048524 <getline>:
2 8048524: 55 push %ebp
3 8048525: 89 e5 mov %esp,%ebp
4 8048527: 83 ec 10 sub $0x10,%esp
5 804852a: 56 push %ecx
6 804852b: 53 push %edi
Diagram stack at this point
7 804852c: 83 c4 f4 add $0xfffffff4,%esp
8 804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
9 8048532: 53 push %ebx
10 8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (2) Modify your diagram to show the effect of the call to scanf (line 10) on the part of the stack shown.

Homework (Part VI)

```
1 08048524 <getline>:
2 8048524: 55 push %ebp
3 8048525: 89 e5 mov %esp,%ebp
4 8048527: 83 ec 10 sub $0x10,%esp
5 804852a: 56 push %ecx
6 804852b: 53 push %edi
Diagram stack at this point
7 804852c: 83 c4 f4 add $0xfffffff4,%esp
8 804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
9 8048532: 53 push %ebx
10 8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

```
+-------------+
| 08 00 33 32 | Return Address
+-------------+
| 31 30 39 38 | saved %ebp
+-------------+
| 37 36 35 34 | buf[4-7]
+-------------+
| 33 32 31 30 | buf[0-3]
+-------------+
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (2) Modify your diagram to show the effect of the call to scanf (line 10) on the part of the stack shown.

Homework (Part VI)

```
+-------------+
| 08 00 33 32 | Return Address
+-------------+
| 31 30 39 38 | saved %ebp
+-------------+
| 37 36 35 34 | buf[4-7]
+-------------+
| 33 32 31 30 | buf[0-3]
+-------------+
```

```
08048524 <getline>:
8048524: 55 push %ebp
8048525: 89 e5 mov %esp,%ebp
8048527: 83 ec 10 sub $0x10,%esp
804852a: 56 push %ecx
804852b: 53 push %edi
Diagram stack at this point
804852c: 83 c4 f4 add $0xfffffff4,%esp
804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
8048532: 53 push %ebx
8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (3) To what address does the program attempt to return (when getline completes)?

Homework (Part VI)

```
+-------------+
| 08 00 33 32 | Return Address
+-------------+
| 31 30 39 38 | saved %ebp
+-------------+
| 37 36 35 34 | buf[4-7]
+-------------+
| 33 32 31 30 | buf[0-3]
+-------------+
```

```
08048524 <getline>:
8048524: 55 push %ebp
8048525: 89 e5 mov %esp,%ebp
8048527: 83 ec 10 sub $0x10,%esp
804852a: 56 push %ecx
804852b: 53 push %edi
Diagram stack at this point
804852c: 83 c4 f4 add $0xfffffff4,%esp
804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
8048532: 53 push %ebx
8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (3) To what address does the program attempt to return (when getline completes)?
  - 0x8003332

Homework (Part VI)

```
+-------------+
| 08 00 33 32 | Return Address
+-------------+
| 31 30 39 38 | saved %ebp
+-------------+
| 37 36 35 34 | buf[4-7]
+-------------+
| 33 32 31 30 | buf[0-3]
+-------------+
```

```
08048524 <getline>:
8048524: 55 push %ebp
8048525: 89 e5 mov %esp,%ebp
8048527: 83 ec 10 sub $0x10,%esp
804852a: 56 push %ecx
804852b: 53 push %edi
Diagram stack at this point
804852c: 83 c4 f4 add $0xfffffff4,%esp
804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
8048532: 53 push %ebx
8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (4) What register(s) have corrupted value(s) when getline returns?

Homework (Part VI)

```c
char *getline()
{
    char buf[8];
    char *result;
    scanf("%s", buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

```
08048524 <getline>:
8048524: 55 push %ebp
8048525: 89 e5 mov %esp,%ebp
8048527: 83 ec 10 sub $0x10,%esp
804852a: 56 push %ecx
804852b: 53 push %edi
Diagram stack at this point
804852c: 83 c4 f4 add $0xfffffff4,%esp
804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
8048532: 53 push %ebx
8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string "01234567890123".
  - (4) What register(s) have corrupted value(s) when getline returns?
  - The saved value of register %ebp was changed to 0x31303938, and this will be loaded into %ebp before getline returns. %eip is corrupted because the return of getline() will effectively pop the corrupted return address into %eip.

Homework (Part VI)

```
char *getline()
{
    char buf[8];
    char *result;
    scanf("%s", buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

```
08048524 <getline>:
8048524: 55 push %ebp
8048525: 89 e5 mov %esp,%ebp
8048527: 83 ec 10 sub $0x10,%esp
804852a: 56 push %ecx
804852b: 53 push %edi
Diagram stack at this point
804852c: 83 c4 f4 add $0xfffffff4,%esp
804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
8048532: 53 push %ebx
8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string "01234567890123".
  - (5) Besides the potential for buffer overflow, what two other things are wrong with the code for getline?

Homework (Part VI)

```c
char *getline()
{
    char buf[8];
    char *result;
    scanf("%s", buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

```
08048524 <getline>:
8048524: 55 push %ebp
8048525: 89 e5 mov %esp,%ebp
8048527: 83 ec 10 sub $0x10,%esp
804852a: 56 push %ecx
804852b: 53 push %edi
Diagram stack at this point
804852c: 83 c4 f4 add $0xfffffff4,%esp
804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
8048532: 53 push %ebx
8048533: e8 74 fe ff ff call 80483ac <_init+0
Modify diagram to show values at this point
```

- Procedure getline is called with the return address equal to 0x804ab8c, register %ebp equal to 0xbffffa60, register %edi equal to 0x3, and register %ecx equal to 0x8. You type in the string “01234567890123”.
  - (5) Besides the potential for buffer overflow, what two other things are wrong with the code for getline?
  - The call to malloc should have had strlen(buf)+1 as its argument, and it should also check that the returned value is non-null.

Quiz (ROP #1)

- a_1: pop ebx; ret
- a_2: pop eax; ret
- a_3: mov eax, (ebx); ret
- a_4: mov ebx, (eax); ret
- a_5: add eax, (ebx); ret
- a_6: push ebx; ret
- a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value 0xBBBBBBB+1 into address 0xAAAAAAA**

Quiz (ROP #1)

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value 0xBBBBBBB+1 into address 0xAAAAAAA**

A2 | 0x1 | A1 | 0xA | A3 | A2 | 0xB | A5 |

low high

Quiz (ROP #2)

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value 0xBBBBBBB+1 into address 0xAAAAAAA** – **then execute from 0xBBBBBBB+1**

Quiz (ROP #2)

a_1: pop ebx; ret

a_2: pop eax; ret

a_3: mov eax, (ebx); ret

a_4: mov ebx, (eax); ret

a_5: add eax, (ebx); ret

a_6: push ebx; ret

a_7: pop esp; ret

Known Gadgets

Draw a stack diagram for a ROP exploit to **store the value 0xBBBBBBB+1 into address 0xAAAAAAA – then execute from 0xBBBBBBB+1**

A2 | 0x1 | A1 | 0xA | A3 | A2 | 0xB | A5 | A7 | 0xA

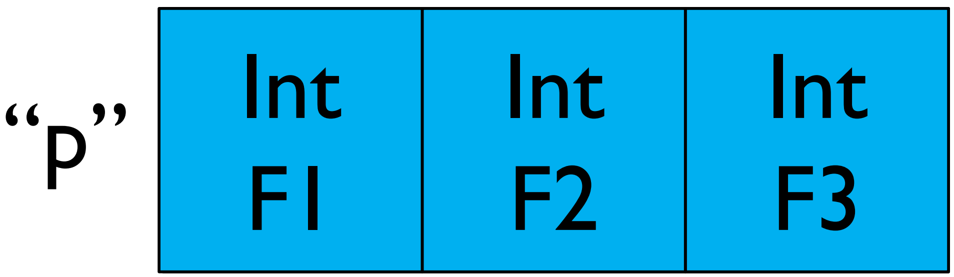
low high

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

Exploiting Type Errors

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

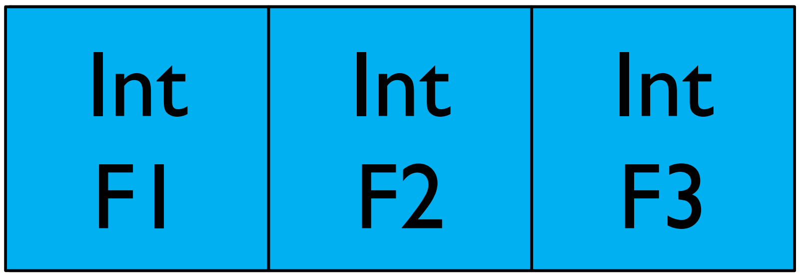


- Only memory large enough for t1 is allocated

Exploiting Type Errors

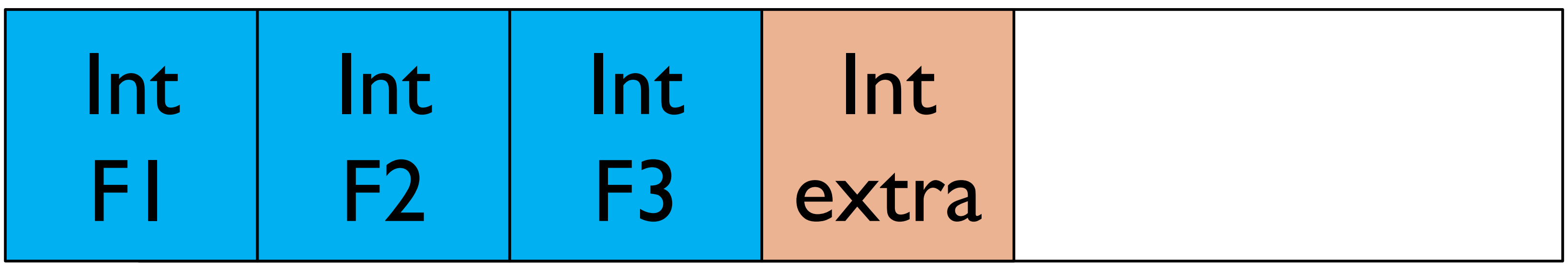
- “p” is assigned to an object of type t1

“p”



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

“q”



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

Memory Error Defenses

- We have discussed some
  - Canaries
  - Address Space Layout Randomization
  - Data Execution Protection (No Execute)
- How do these defenses work? Review

Memory Error Defenses

- We have discussed some
  - Canaries
  - Address Space Layout Randomization
  - Data Execution Protection (No Execute)
- These defenses do not prevent ROP attacks
  - Why not?

Memory Error Defenses

- We have discussed some
  - Canaries
  - Address Space Layout Randomization
  - Data Execution Protection (No Execute)
- These defenses do not prevent ROP attacks
  - Why not?
    - Bypass canaries and ASLR
      - Disclose canary values on stack
      - Disclose stack pointer values (EBP)
    - DEP/NX does not prevent execution of code memory

Conclusions

- Structure of exam
  - Multiple choice – fill in blank
  - Short answer – Conceptual questions
    - May be more than one question – be sure to answer all
  - Constructions – Problem solving
    - Multiple sub-parts
- Time management – answer ones you know
- Topics – Covered in these slides
  - Those in this review may be on the exam (up to ROP)
- Readings – good to know more – different angle

Questions