Assignment 2: Sploits 3 and 4

CSE 127 Week 3 DI 4/19/19

NOTE

Assignment 2 is due on 5/3, Friday of Week 5.

The midterm is on 4/30, Tuesday of Week 5.

Try to Finish Assignment 2 before the midterm

All of it (and more) is fair game for the midterm

Reminder: The Setting

- target[1-4].c are vulnerable pieces of code that each read a string from the command line
- Our exploit is the string we pass in
- We could run the attack by running $ ./target1 “attack_string_here”
- But that’s hard
  - Hard to type the string and fix things at specific locations
  - Some of the strings may be really long
- So we call our targets from C programs called sploit[1-4].c
- Just think of sploit[1-4].c as the C version of calling ./target from the shell
- **You only get to modify sploit[1-4].c. You CAN’T change the target**

The Setting

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

#define TARGET "/tmp/target1"

int main(void)
{
  char *args[3];
  char *env[1];

  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  if (0 > execve(TARGET, args, env))
    fprintf(stderr, "execve failed.\n");

  return 0;
}
```

sploit1.c

Refer to last Week's discussion slides, lecture slides, and the Aleph One paper for more

Questions about Sploits 1 or 2?

Last week’s discussion slides can be found on Piazza under Resources.

Take a minute to go through them, and feel free to ask questions

Review

1) What is ebp + 4?
2) What is ***stored at*** ebp + 4?
3) What is stored at ebp?
4) What’s the difference in vulnerabilities between target1 and target2?

```
int bar(char *arg, char *out)
{
  strcpy(out, arg);
  return 0;
}

int foo(char *argv[])
{
  char buf[768];
  bar(argv[1], buf);
}
```

```
void nstrcpy(char *out, int outl, char *in)
{
  int i, len;

  len = strlen(in);
  if (len > outl)
    len = outl;

  for (i = 0; i <= len; i++)
    out[i] = in[i];
}
```

Review

5) What control data (return address, frame pointer, pc, stack pointer, etc) is being corrupted in sploit1?

6) What control data is being corrupted in sploit2?

7) How do you know if your exploit has succeeded?

8) True/False: When you want to quit or exit VirtualBox, you should first power off the VM.

Sploit 3

Target3: The vulnerability

Input Format:

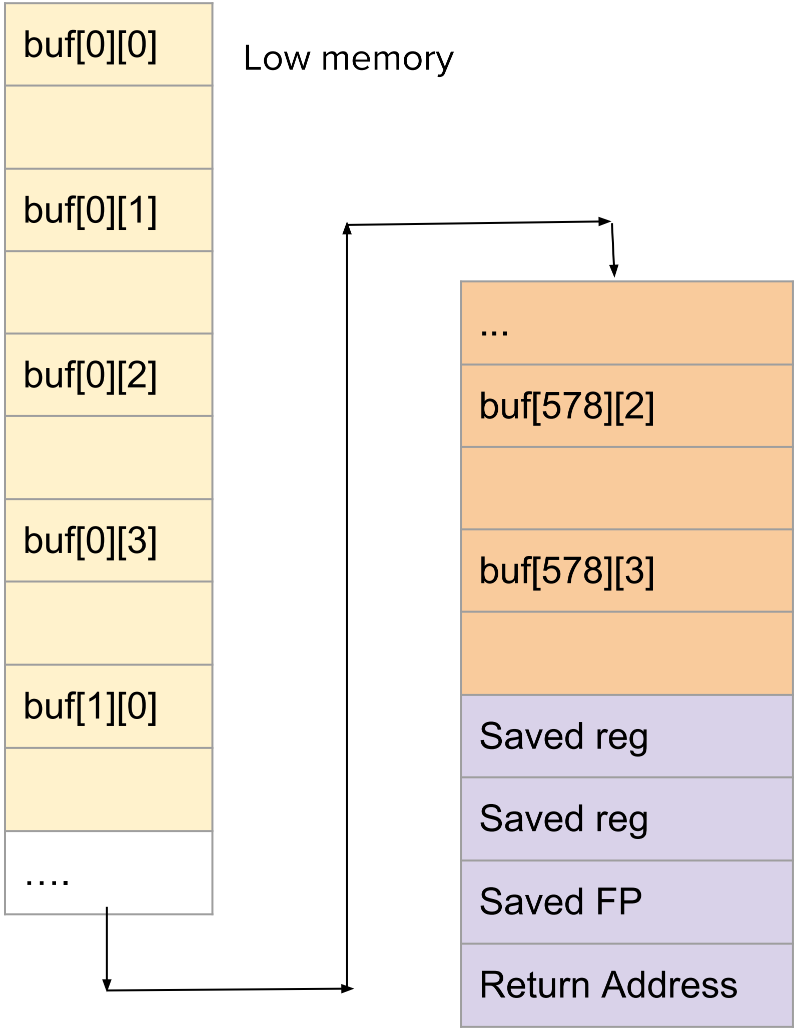
./target3 421,abcdefg....

foo(“abcdefg...”, 421)

```
struct widget_t {
  double x[4];
};

int foo(char *in, int count)
{
  struct widget_t buf[579];

  if (count < 579)
    memcpy(buf, in, count * sizeof(struct widget_t));

  return 0;
}
```

```
int main(int argc, char *argv[])
{
  int count;
  char *in;

  if (argc != 2)
    {
      fprintf(stderr, "target3: argc != 2\n");
      exit(EXIT_FAILURE);
    }

  /*
   * format of argv[1] is as follows:
   *
   * - a count, encoded as a decimal number in ASCII
   * - a comma (",")
   * - the remainder of the data, treated as an array
   *   of struct widget_t
   */

  count = (int)strtoul(argv[1], &in, 10);
  if (*in != ',')
    {
      fprintf(stderr, "target3: argument format is [count],[data]\n");
      exit(EXIT_FAILURE);
    }
  in++;                          /* advance one byte, past the comma */
  foo(in, count);

  return 0;
}
```

Low memory



High memory

```
struct widget_t {
  double x[4];
};

int foo(char *in, int count)
{
  struct widget_t buf[579];

  if (count < 579)
    memcpy(buf, in, count * sizeof(struct widget_t));

  return 0;
}
```

With ‘count’ we control how many widgets are copied.

Can we copy more than 579 widgets? How do we pass the bounds check?

Steps

1) Pass the bounds check
2) Multiply (count * sizeof(struct widget_t))
3) Overflow to change the return address

```
struct widget_t {
  double x[4];
};

int foo(char *in, int count)
{
  struct widget_t buf[579];

  if (count < 579)
    memcpy(buf, in, count * sizeof(struct widget_t));

  return 0;
}
```

1) Bounds Check -- Types

./target3 “579,<malicious stuff>”

count = (int) stroul(argv[1] ....)

argv[1] is a **string** (char *) -- “579,<malicious stuff>”

return value of stroul is an **unsigned long** -- 0x00000243

count is a **signed int** -- 0x00000243

* For 579 the 2 representations are the same, but ...

Signed vs unsigned ints/longs

Signed: MSb is the sign bit (0 means positive 1 means negative)

Unsigned: MSb is just the bit for the highest place value

Example: 0xFFFFFFFF (0xF = 1111)

Signed: -1

Unsigned: 4294967295

```
struct widget_t {
  double x[4];
};

int foo(char *in, int count)
{
  struct widget_t buf[579];

  if (count < 579)
    memcpy(buf, in, count * sizeof(struct widget_t));

  return 0;
}
```

* Which of the two above could get compared in the bounds check?

2) Multiplication

sizeof(struct widget_t) = 4 * 8 = 32 = 2^5

x * 2^n == x << n (multiplication vs shift)

Example: 9 * 32 = 9 << 5

9 = 0b0000000.....1001

<< 5 = 0b00.....100100000

= 1(32) + 1(256) = 288

```
struct widget_t {
  double x[4];
};

int foo(char *in, int count)
{
  struct widget_t buf[579];

  if (count < 579)
    memcpy(buf, in, count * sizeof(struct widget_t));

  return 0;
}
```

2) Multiplication

sizeof(struct widget_t) = 4 * 8 = 32 = 2^5

x * 2^n == x << n (mutiplication vs shift)

Example: 9 * 32 = 12 << 5

12 = 0b0000000.....1001

<< 5 = 0b00.....100100000

= 1(32) + 1(256) = 288

Another Example:

4160749577 * 32 = 4160749577 << 5

4160749577 =

0b1111100.....1001

<< 5 = 0b00.....100100000

= 1(32) + 1(256) = 288

2) Multiplication

sizeof(struct widget_t) = 4 * 8 = 32 = 2^5

x * 2^n == x << n (mutiplication vs shift)

Example: 9 * 32 = 12 << 5

12 = 0b0000000.....1001

<< 5 = 0b00.....100100000

= 1(32) + 1(256) = 288

Another Example:

4160749577 * 32 = 4160749577 << 5

4160749577 =

0b1111100.....1001

<< 5 = 0b00.....100100000

= 1(32) + 1(256) = 288

Notice the MSb!

Unsigned: 4160749577

Signed: -134217719

-134217719 * 32 = 288

3) Exploiting this

Take the number you really want — 580

Disguise it as negative — 580 | 0x8000000 = 134218308

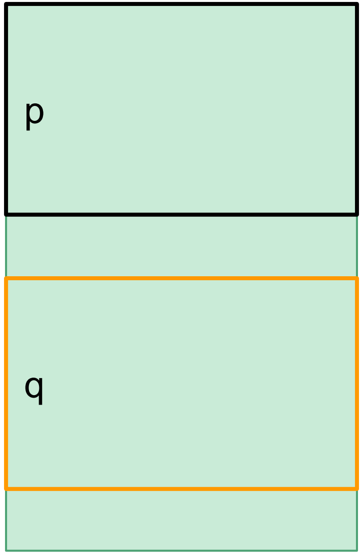
Let the multiplication unmask the disguise — 134218308 * 32 = 580 * 32

./target3 “134218308,\x90\x90\x90<shellcode>....”

Note: Remember that everything before the comma won’t be part of the string ‘in’. You may need some padding (NOPs) to make sure your address aligns correctly.

Sploit 4

Target4: Find the Vulnerability

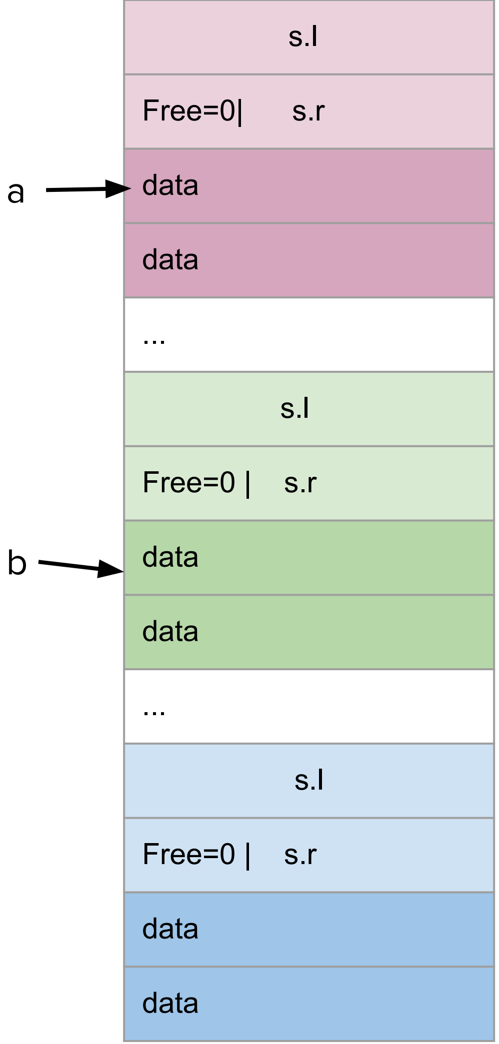



```
int foo(char *arg)
{
  char *p;
  char *q;

  if ( (p = tmalloc(300)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (q = tmalloc(325)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(p);
  tfree(q);

  if ( (p = tmalloc(1024)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(p, arg, 1024);

  tfree(q);

  return 0;
}
```

Heap Chunks

a = malloc(...)

b = malloc(...)

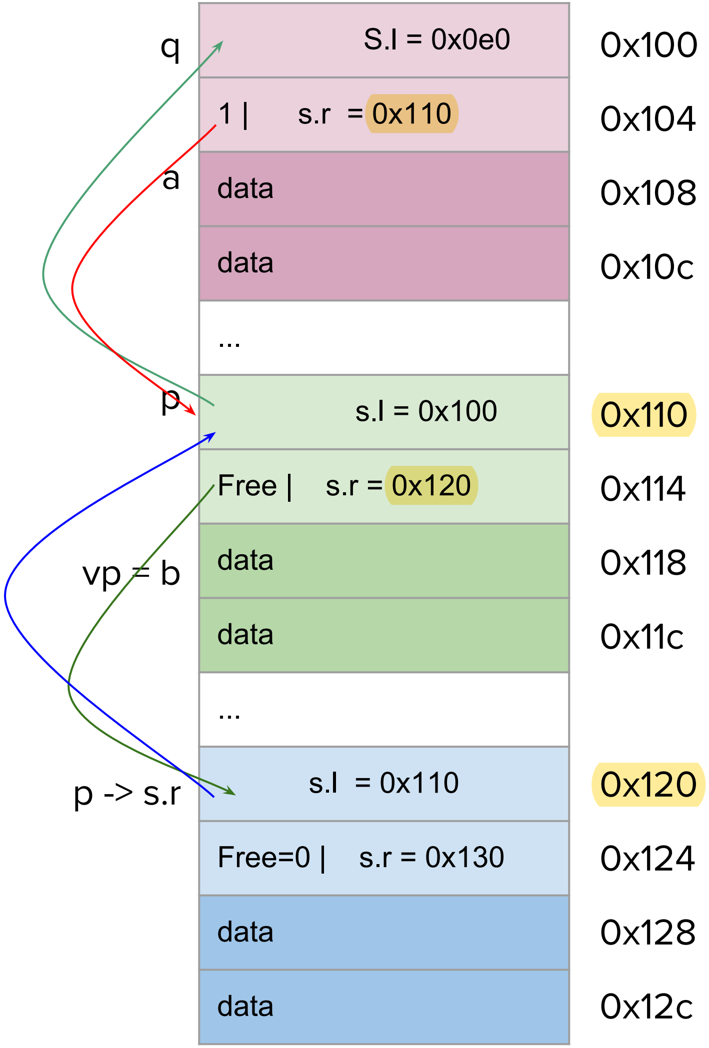


tfree()

Assume a was already freed, and now we're calling tfree(b)

coalesce leftward...

```c
void tfree(void *vp)
{
  CHUNK *p, *q;

  if (vp == NULL)
    return;

  p = TOCHUNK(vp);
  CLR_FREEBIT(p);
  q = p->s.l;
  if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
    {
      CLR_FREEBIT(q);
      q->s.r       = p->s.r;
      p->s.r->s.l = q;
      SET_FREEBIT(q);
      p = q;
    }
```

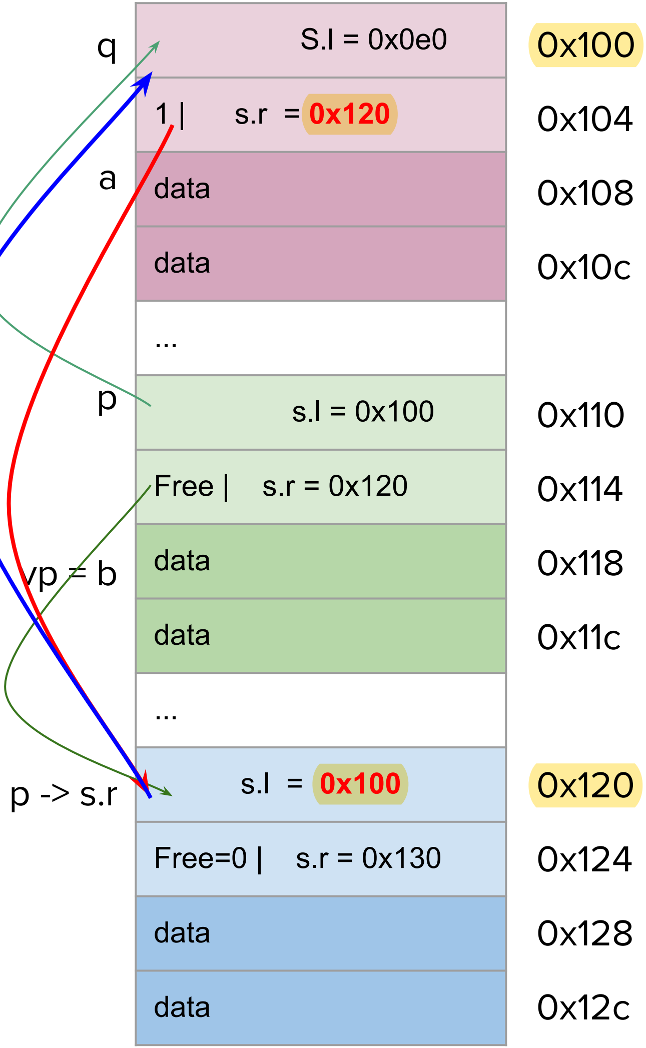


tfree()

Assume a was already freed, and now we’re calling tfree(b)

coalesce leftward...

```c
void tfree(void *vp)
{
  CHUNK *p, *q;

  if (vp == NULL)
    return;

  p = TOCHUNK(vp);
  CLR_FREEBIT(p);
  q = p->s.l;
  if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
    {
      CLR_FREEBIT(q);
      q->s.r      = p->s.r;
      p->s.r->s.l = q;
      SET_FREEBIT(q);
      p = q;
    }
```



Aside: structs and memory

```
struct foo{
        int a;
        int b;
};
struct bar {
        struct foo * p1;
        struct foo * p2;
};
struct bar * s1 = malloc(sizeof(struct bar));
struct foo * s2 = malloc(sizeof(struct foo));
s1 -> p2 = s2;
s1 ->p2->a = 5;
```

s1 →	s1.p1	0x100
	s1.p2 = 0x10c	0x104
		0x108
s2 →	s2.a = 5	0x10c
	s2.b	0x110
	….	….
	s2 = 0x10c	0xff8
	s1 = 0x100	0xffc

Aside: structs and memory

```
struct foo{
        int a;
        int b;
};
struct bar {
        struct foo * p1;
        struct foo * p2;
};
struct bar * s1 = malloc(sizeof(struct bar));
struct foo * s2 = malloc(sizeof(struct foo));
s1 -> p2 = s2;
s1 ->p2->a = 5;
```

Equivalent to: *(s1 + 4 bytes) = s2
*(*(s1 + 4) + 0) = 5

Heap

s1 →	s1.p1	0x100
	s1.p2 = 0x10c	0x104
		0x108
s2 →	s2.a = 5	0x10c
	s2.b	0x110
	….	….
	s2 = 0x10c	0xff8
	s1 = 0x100	0xffc

Stack

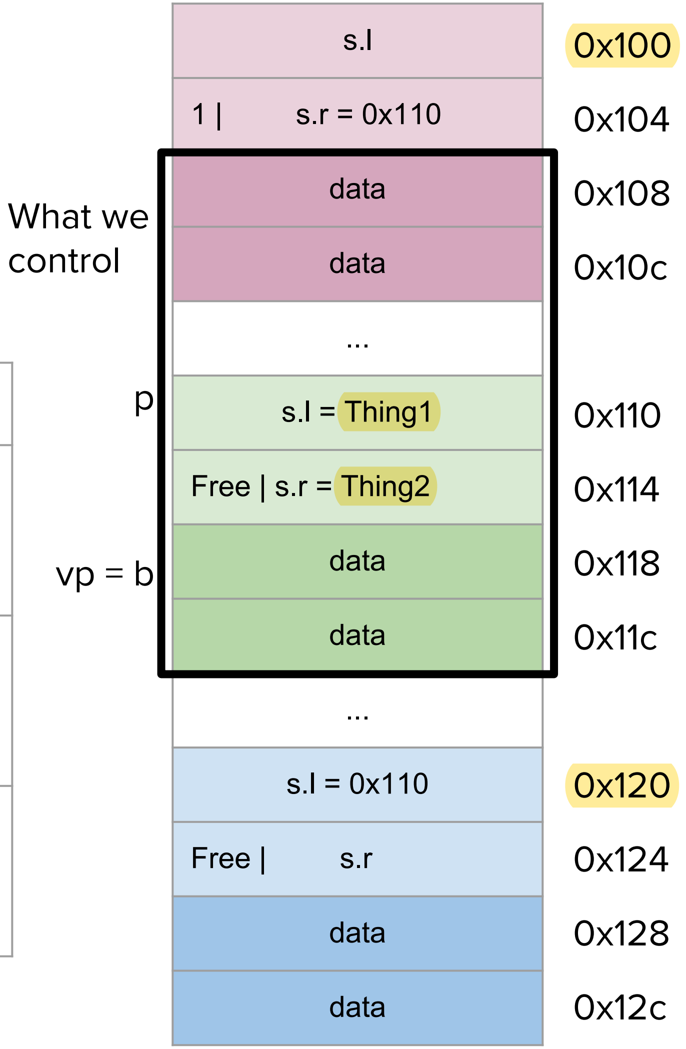
CHUNK struct

Linked List Code	Arbitrary Pointer Operations
q = p -> s.l	q = *(p + 0) q = *p
q->s.r = p -> s.r	*(q + 4) = *(p + 4) *(*p + 4) = *(p + 4)
p->s.r->s.l = q	*(*(p+4) + 0) = q *(*(p+4)) = *p

```
typedef double ALIGN;

typedef union CHUNK_TAG
{
  struct
    {
      union CHUNK_TAG *l;
      union CHUNK_TAG *r;
    } s;
  ALIGN x;
} CHUNK;
```

What memory will free() change?

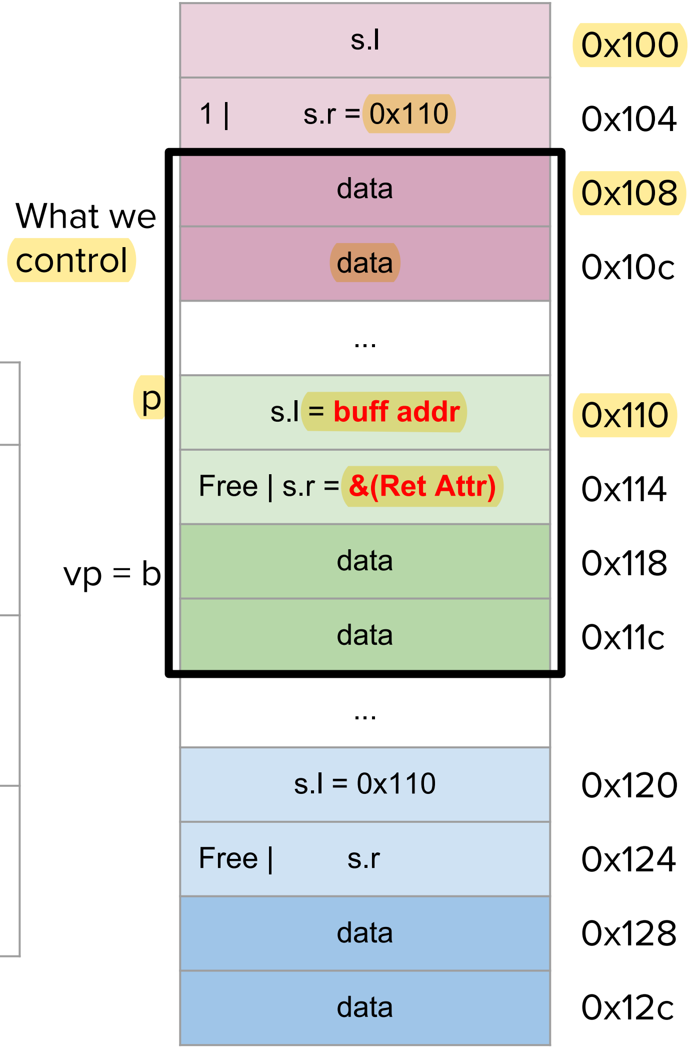
Linked List Code	Arbitrary Pointer Operations
q = p -> s.l	q = *(p + 0) **q = Thing1**
q->s.r = p -> s.r	*(q + 4) = *(p + 4) **Thing1[4-7] = Thing2**
p->s.r->s.l = q	*(*(p+4) + 0) = q **Thing2[0-3] = Thing1**



So what if in that memory we put…

The address of the Ret Addr is $ebp +4

Linked List Code	Arbitrary Pointer Operations
q = p -> s.l	q = *(p + 0) **q** = **buf**
q->s.r = p -> s.r	*(q + 4) = *(p + 4) **buf[4-7] = &(ret addr)**
p->s.r->s.l = q	*(*(p+4) + 0) = q **Ret addr = buf**



Free Bit

In order to enter the if, we must pass GET_FREEBIT(q), so the LSb of q->s.r needs to be 1

- Remember Little Endian
- the CLR_FREEBIT operation makes sure that the original q->s.r is used for the coalescing, and SET_FREEBIT sets the bit back.

```
void tfree(void *vp)
{
  CHUNK *p, *q;

  if (vp == NULL)
    return;

  p = TOCHUNK(vp);
  CLR_FREEBIT(p);
  q = p->s.l;
  if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
    {
      CLR_FREEBIT(q);
      q->s.r      = p->s.r;
      p->s.r->s.l = q;
      SET_FREEBIT(q);
      p = q;
    }
```

Breakdown

Linked List Code	Arbitrary Pointer Operations	**Exploit result**
q = p -> s.l	q = *(p + 0) q = *p	**q = &buf**
q->s.r = p -> s.r	*(q + 4) = *(p + 4) *(*p + 4) = *(p + 4)	**buf[4-7] = ret addr**
p->s.r->s.l = q	*(*(p+4) + 0) = q *(*(p+4)) = *p	**ret addr = &buf**

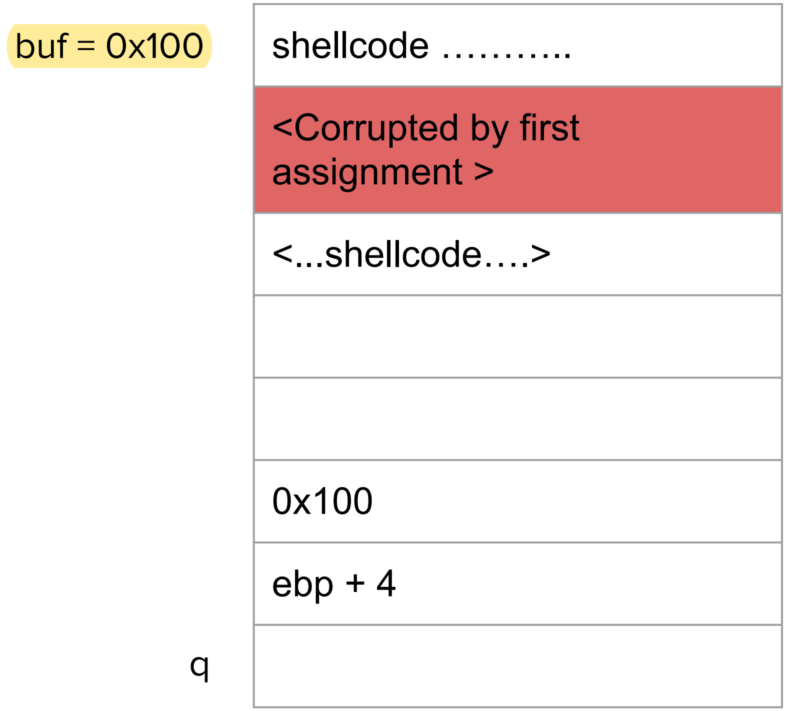
Side Effect: Corruption

buf[4-7] = &(ret addr)

- Corrupts our buffer

Ret addr = buf

- What we want



Side Effect: Corruption

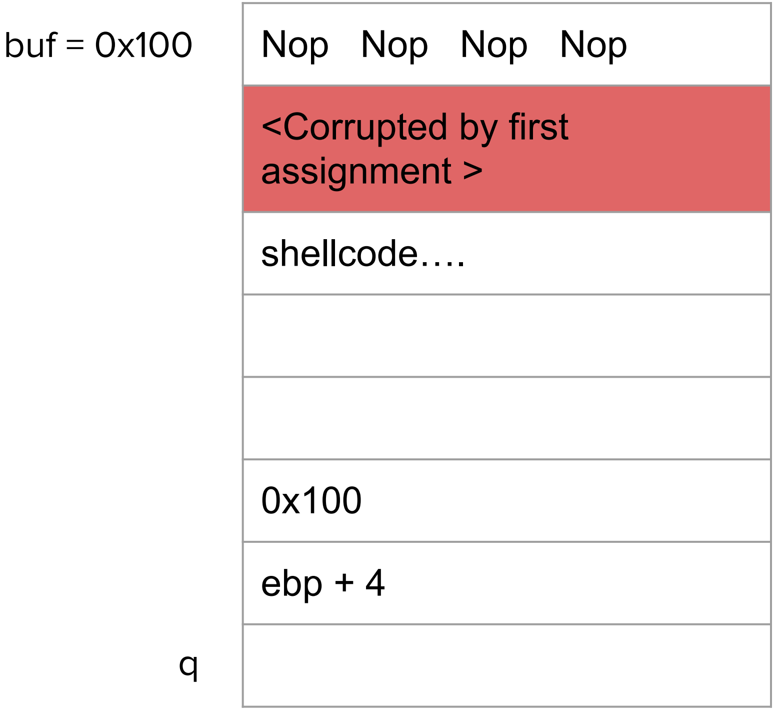
buf[4-7] = &(ret addr)

- Corrupts our buffer

Ret addr = buf

- What we want

“Solution” 1: Nops?



Side Effect: Corruption

(0x108)[4-7] = &(ret addr)

- Corrupts our buffer

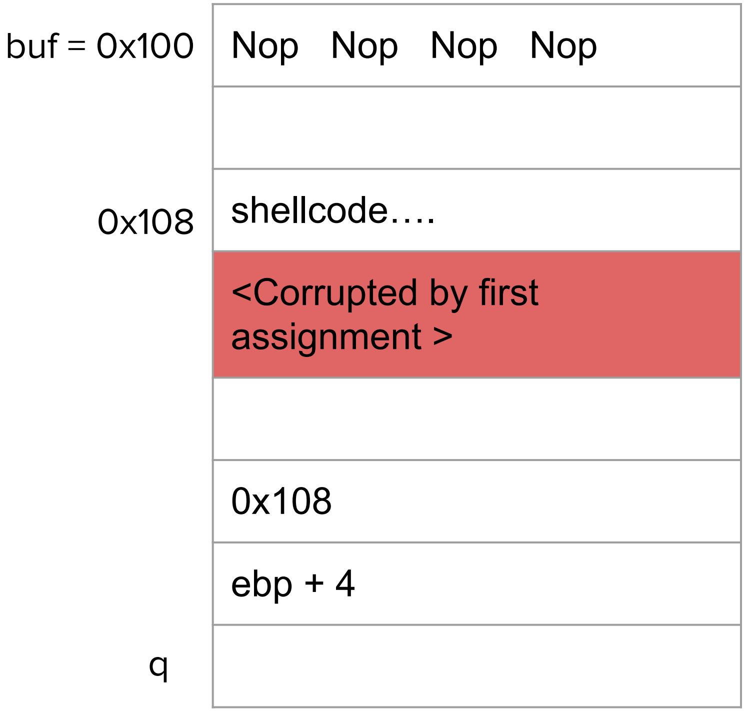
Ret addr = buf

- What we want

“Solution” 1: Nops?

- Still execute corrupted address

“Solution” 2: Choose a later address?



Side Effect: Corruption

(0x108)[4-7] = &(ret addr)

- Corrupts our buffer

Ret addr = buf

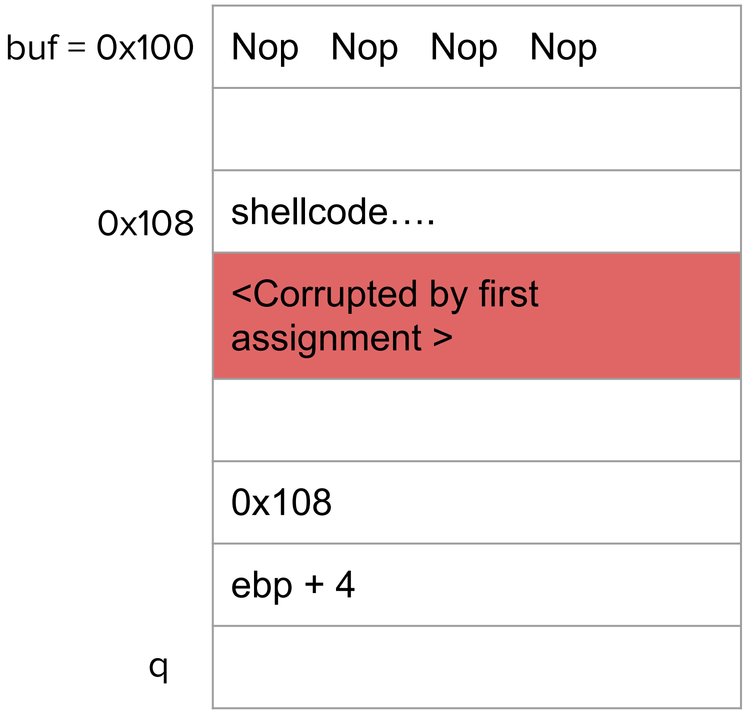
- What we want

“Solution” 1: Nops?

- Still execute corrupted address

“Solution” 2: Choose a later address?

- The corruption moves with us



Jump the corruption

buf[4-7] = &(ret addr)

- Corrupts our buffer

Ret addr = buf

- What we want

Solution 3: Jmp over the corrupted memory

JMP instruction (JMP rel16/32)

- http://ref.x86asm.net/coder32.html

How much to jump?

- Relative to the first byte after ‘Amt’

buf = 0x100

JMP Amt NOP NOP
<Corrupted by first assignment >
shellcode….
0x100
ebp + 4

q

How do we fix these vulnerabilities?

1. Buffer overflow
2. Buffer overflow (off by 1)
3. Integer
4. Double free()

Questions?