Software Security

Low-level Vulnerabilities

- Programs written in **C and C++** are susceptible a variety of dangerous **vulnerabilities**
  - **Buffer overflows**
    - On the stack
    - On the heap
    - Due to integer overflow
    - Over-writing and over-reading
  - **Format string mismatches**
  - **Dangling pointer dereferences**

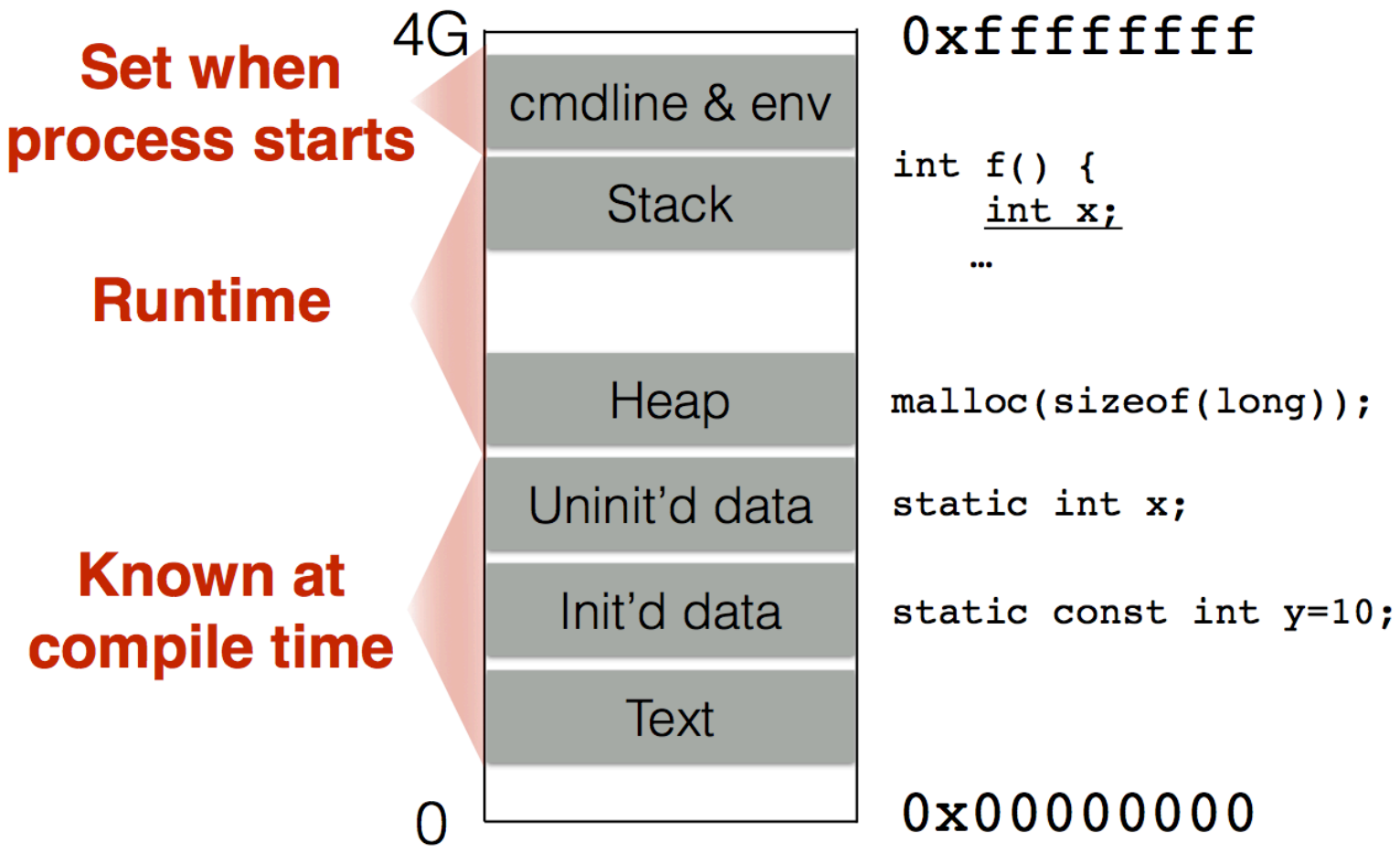
Attacks

- *Stack smashing*
- *Format string attack*
- *Stale memory access*
- *Return-oriented Programming (ROP)*

- All **violations** of **memory safety**
  - Accesses to memory via pointers that don't *own* that memory

Memory layout

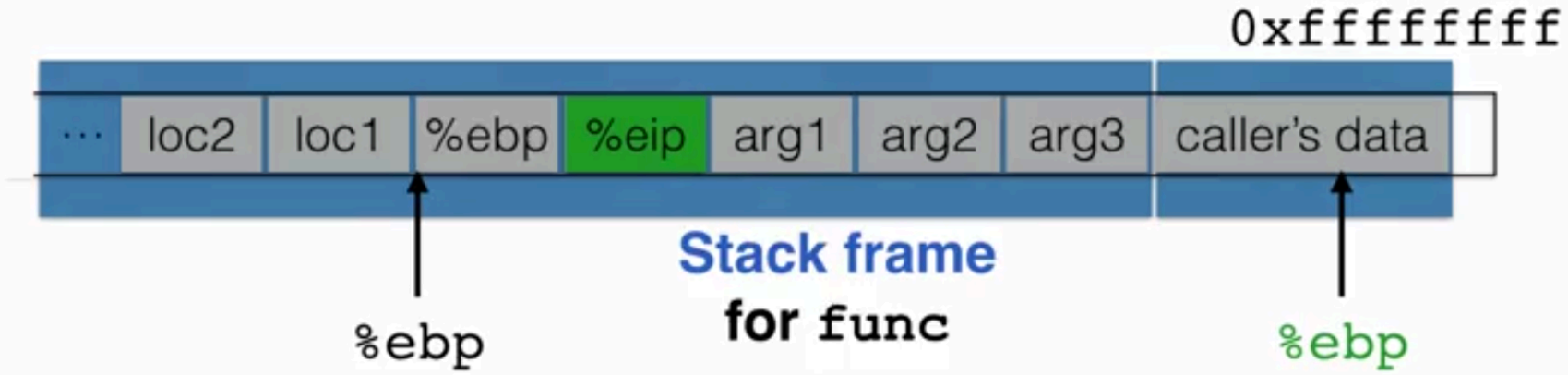
Location of data areas



Returning from functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we resume here?



Set `%eip to 4(%ebp)` at return

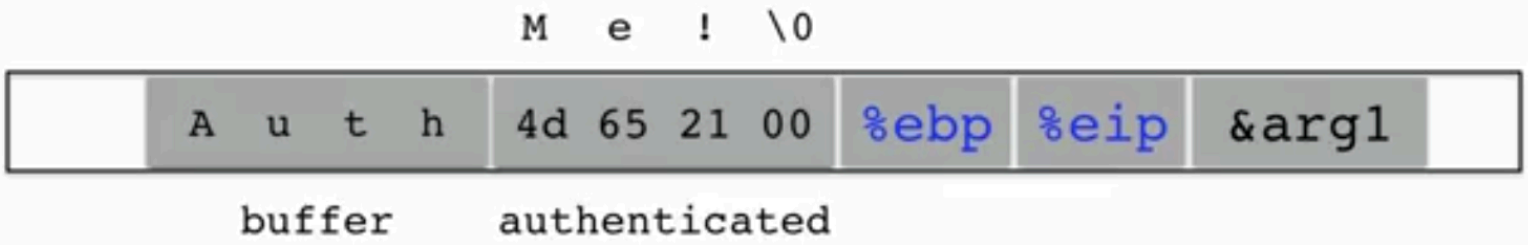
Push next `%eip` before `call`

Buffer overflows

Security-relevant outcome

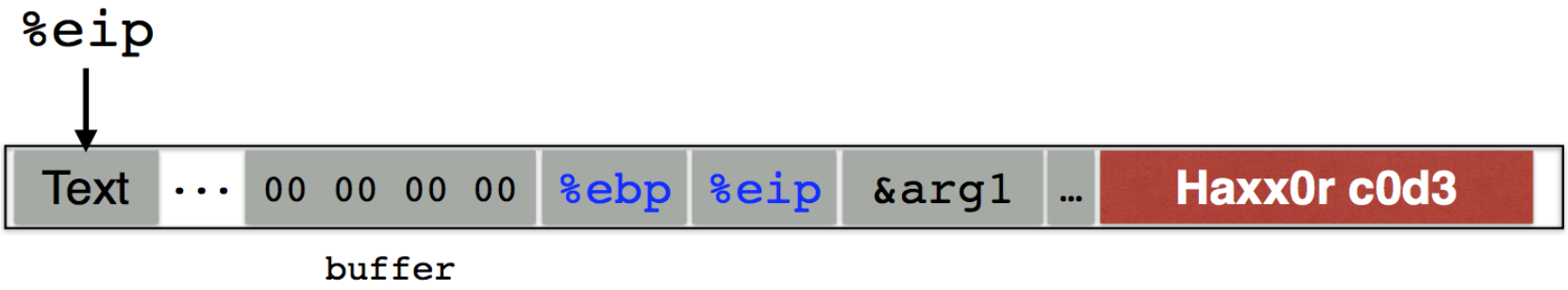
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



Code Injection: Main idea

```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

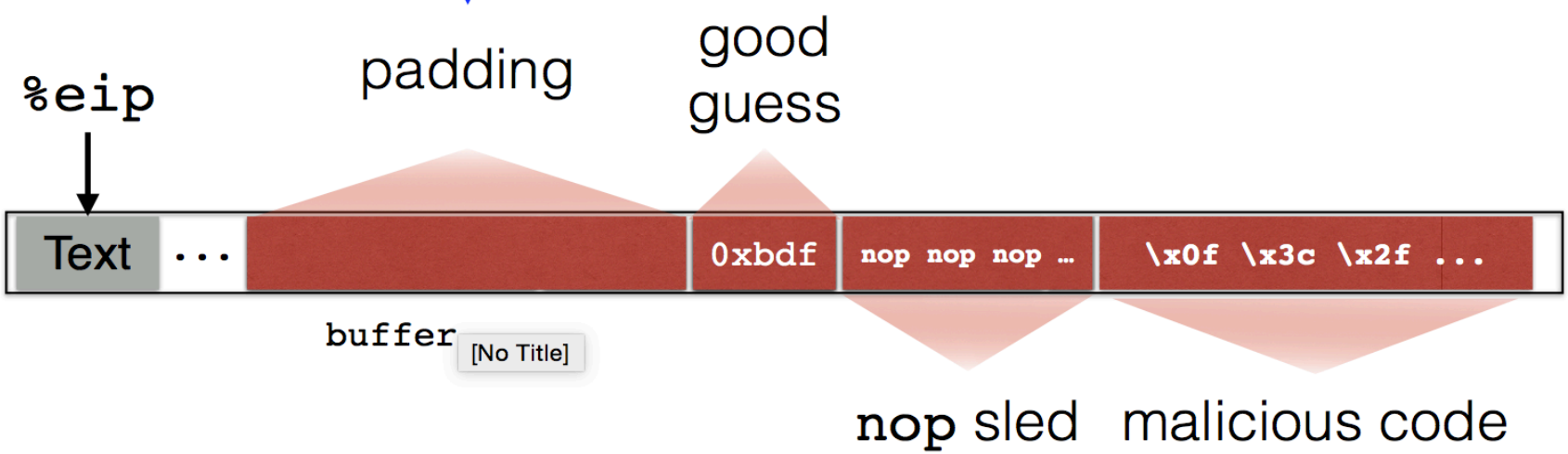


(1) Load my own code into memory

(2) Somehow get `%eip` to point to it

Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to **`gets`**/etc. begins.



Other memory exploits

Heap overflow

- Stack smashing overflows a stack allocated buffer
- You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**

Heap overflow

```
typedef struct _vulnerable_struct {
  char buff[MAX_LEN];
  int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
 strcpy( s->buff, one );
 strcat( s->buff, two );
 return s->cmp( s->buff, "file://foobar" );
}
```

copy ***one*** *into* ***buff***

copy ***two*** *into* ***buff***

must have ***strlen(one)+strlen(two) < MAX_LEN***

or we overwrite s->cmp

Heap overflow variants

- **Overflow into the C++ object *vtable***
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object's methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work
- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap
- **Overflow heap metadata**
  - Hidden header just before the pointer returned by malloc
  - Flow into that header to corrupt the heap itself
    - Malloc implementation to do your dirty work for you!

Integer overflow

```
void vulnerable()
{
  char *response;
  int nresp = packet_get_int();
  if (nresp > 0) {
   response = malloc(nresp*sizeof(char*));
   for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
  }
```

Integer overflow

```
void vulnerable()
{
  char **response;
  int nresp = packet_get_int();
  if (nresp > 0) {
   response = malloc(nresp*sizeof(char*));
   for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
  }
```

HUGE

- If we set `nresp` to 1073741824 and `sizeof(char*)` is 4

Integer overflow

```
void vulnerable()
{
  char **response;
  int nresp = packet_get_int();
  if (nresp > 0) {
   response = malloc(nresp*sizeof(char*));
   for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
  }
```

HUGE

Wrap-around

- If we set `nresp` to 1073741824 and `sizeof(char*)` is 4
- then `nresp*sizeof(char*)` overflows to become 0

Integer overflow

```
void vulnerable()
{
  char **response;
  int nresp = packet_get_int();
  if (nresp > 0) {
   response = malloc(nresp*sizeof(char*));
   for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
  }
```

HUGE

Wrap-around

Overflow

- If we set **`nresp`** to 1073741824 and **`sizeof(char*)`** is 4
- then **`nresp*sizeof(char*)`** overflows to become 0
- subsequent writes to allocated **`response`** overflow it

Corrupting data

- The attacks we have shown so far **affect code**
  - *Return addresses* and *function pointers*

- But attackers can **overflow data** as well, to
  - **Modify a secret key** to be one known to the attacker, to be able to decrypt future intercepted messages
  - **Modify state variables** to bypass authorization checks (earlier example with `authenticated` flag)
  - **Modify interpreted strings** used as part of commands
    - E.g., to facilitate SQL injection, discussed later in the course

Read overflow

- Rather than permitting writing past the end of a buffer, a bug could permit **reading past the end**
- Might **leak secret information**

Read overflow

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   for (i=0; i<len; i++)
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
 }}
```

Read integer

Read message

Echo back (partial) message

May exceed actual message length!

Sample transcript

```
% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|
```

OK: input length < buffer size

BAD: length > size !

leaked data

Heartbleed

- The **Heartbleed bug** was a read overflow in exactly this style
- The SSL server should accept a “heartbeat” message that it echoes back
- The heartbeat message specifies the length of its echo-back portion, but the buggy SSL **software did not check the length was accurate**
- Thus, an attacker could request a longer length, and **read past the contents of the buffer**
  - Leaking passwords, crypto keys, …

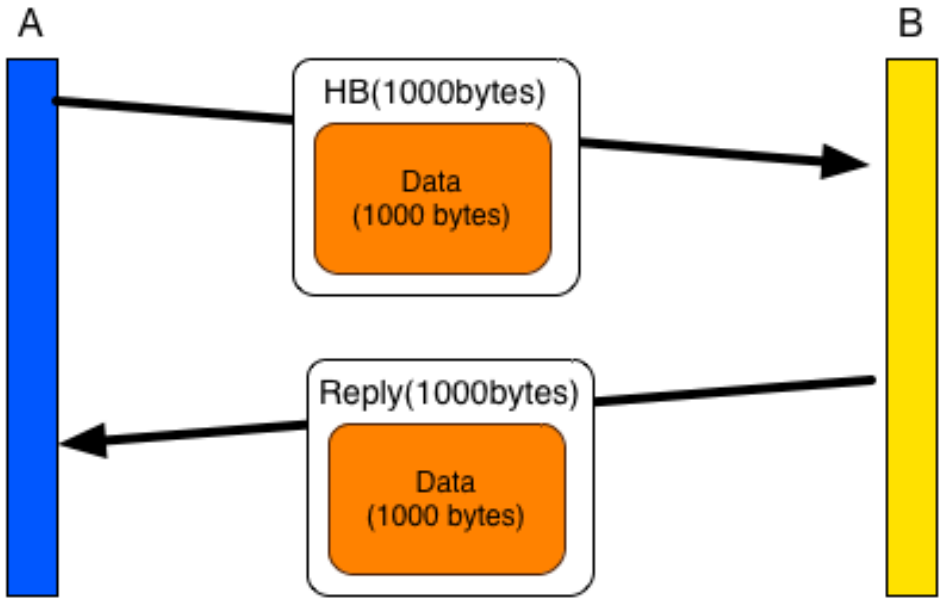
Internet Engineering Task Force (IETF)
Request for Comments: 6520
Category: Standards Track
ISSN: 2070-1721

R. Seggelmann
M. Tuexen
Muenster Univ. of Appl. Sciences
M. Williams
GWhiz Arts & Sciences

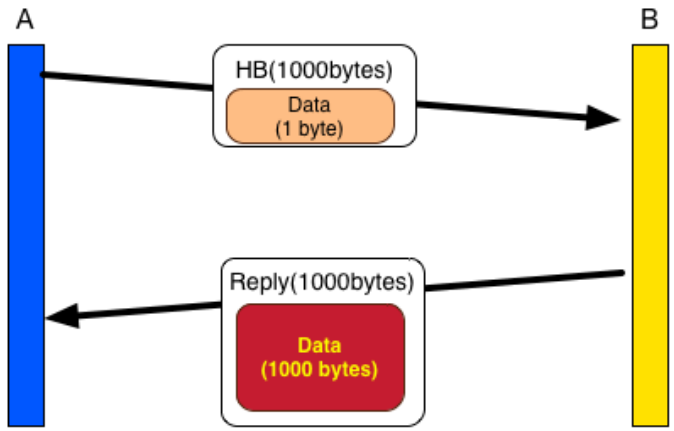
February 2012

Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension

When a HeartbeatRequest message is received and sending a HeartbeatResponse is not prohibited as described elsewhere in this document, the receiver MUST send a corresponding HeartbeatResponse message carrying an exact copy of the payload of the received HeartbeatRequest.



```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
} HeartbeatMessage;
```



Frame (151 bytes) | Decrypted SSL record (40 bytes)

```
0000  01 03 fd 68 65 61 72 74 62 6c 65 65 64 2e 66 69   ...heartbleed.fi
001?  6c 69 70 7? 6f 2e 69 6f 59 45 4c 4c 4f 57 20 53   lippo.ioYELLOW S
0020  55 42 4d 41 52 49 4e 45                           UBMARINE
```

Type Length

Frame (1159 bytes) | Decrypted SSL record (1040 bytes)

```
0000  02 03 fd 68 65 61 72 74 62 6c 65 65 64 2e 66 69   ...heartbleed.fi
0010  6c 69 70 70 6f 2e 69 6f 59 45 4c 4c 4f 57 20 53   lippo.ioYELLOW S
0020  55 42 4d 41 52 49 4e 45 27 19 8d bd c6 d3 3d b3   UBMARINE'.....=.
0030  f6 44 b1 1f fb 61 14 73 0e f4 d1 96 03 03 03 03   .D...a.s........
0040  f4 5d 50 82 82 b5 eb 68 28 53 60 69 fc b6 94 21   .]P....h(S`i...!
0050  a1 7e 7a 68 16 be 9c d6 0f ec d0 b1 00 4c 82 4a   .~zh.........L.J
```

```
0220  4c c4 dd 29 ac c1 ea 1d 1c 20 19 2f b1 68 63 fb   L..).....  ./.hc.
0230  18 3b 6f 53 35 36 63 36 37 0d 0a 0d 0a 5f 6d 65   .;oS56c67...._me
0240  74 68 6f 64 3d 50 55 54 26 6c 6f 67 69 6e 3d 61   thod=PUT&login=a
0250  64 6d 69 6e 26 70 61 73 73 77 6f 72 64 3d 63 6c   dmin&password=cl
0260  6f 75 64 73 68 61 72 6b 23 13 2a 85 7f 17 ad a3   oudshark#.*.....
0270  5c 15 b0 8e d0 94 45 78 00 00 00 00 00 00 00 00   \.....Ex........
```

What's the difference?

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

What's the difference?

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

Attacker controls the format string