Auditing C Code

Dr. Jared DeMott
CTO AND FOUNDER

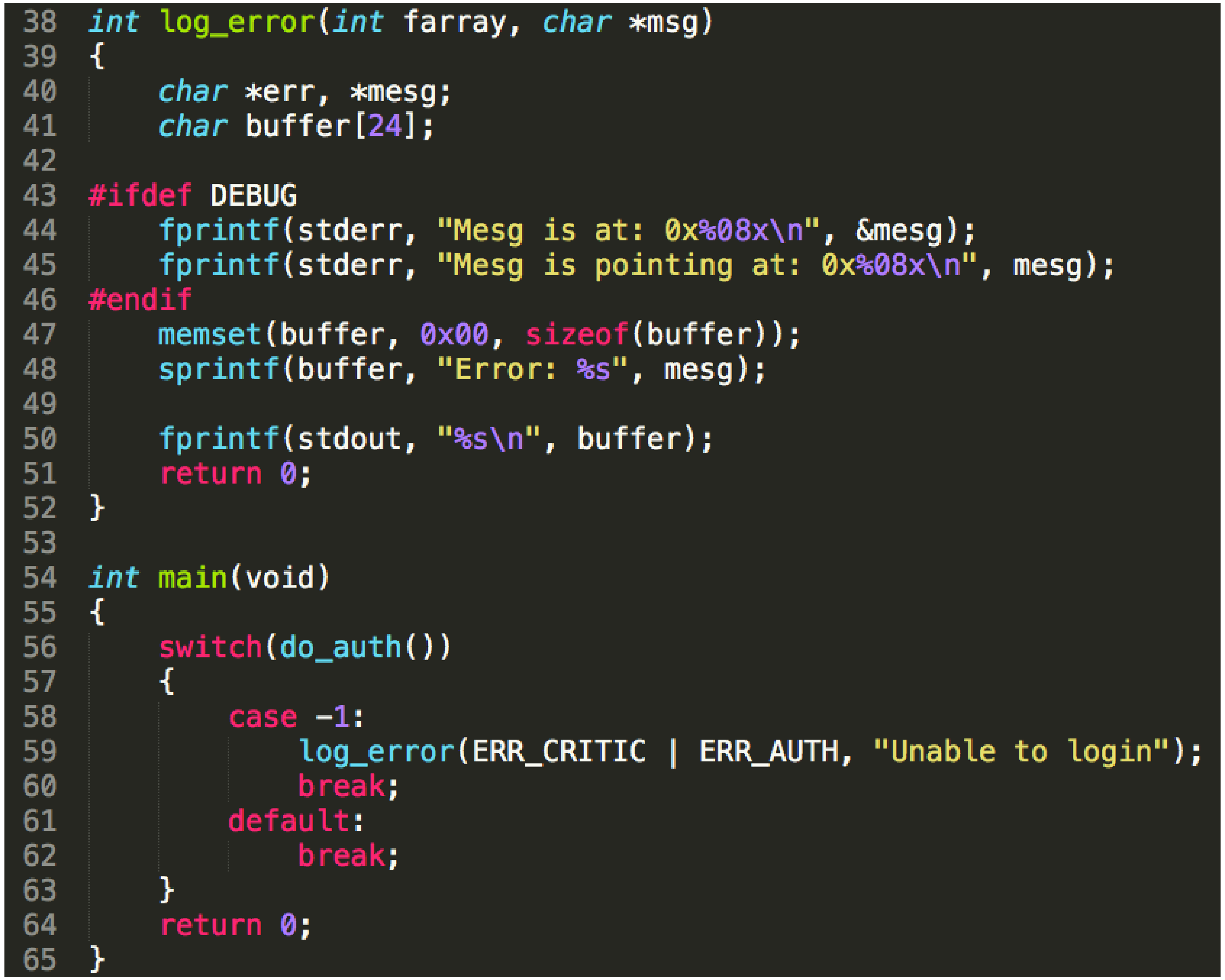
@jareddemott www.vdalabs.com

Introduction

Find common security bugs

Demonstrated by show-and-spot

Heartbleed



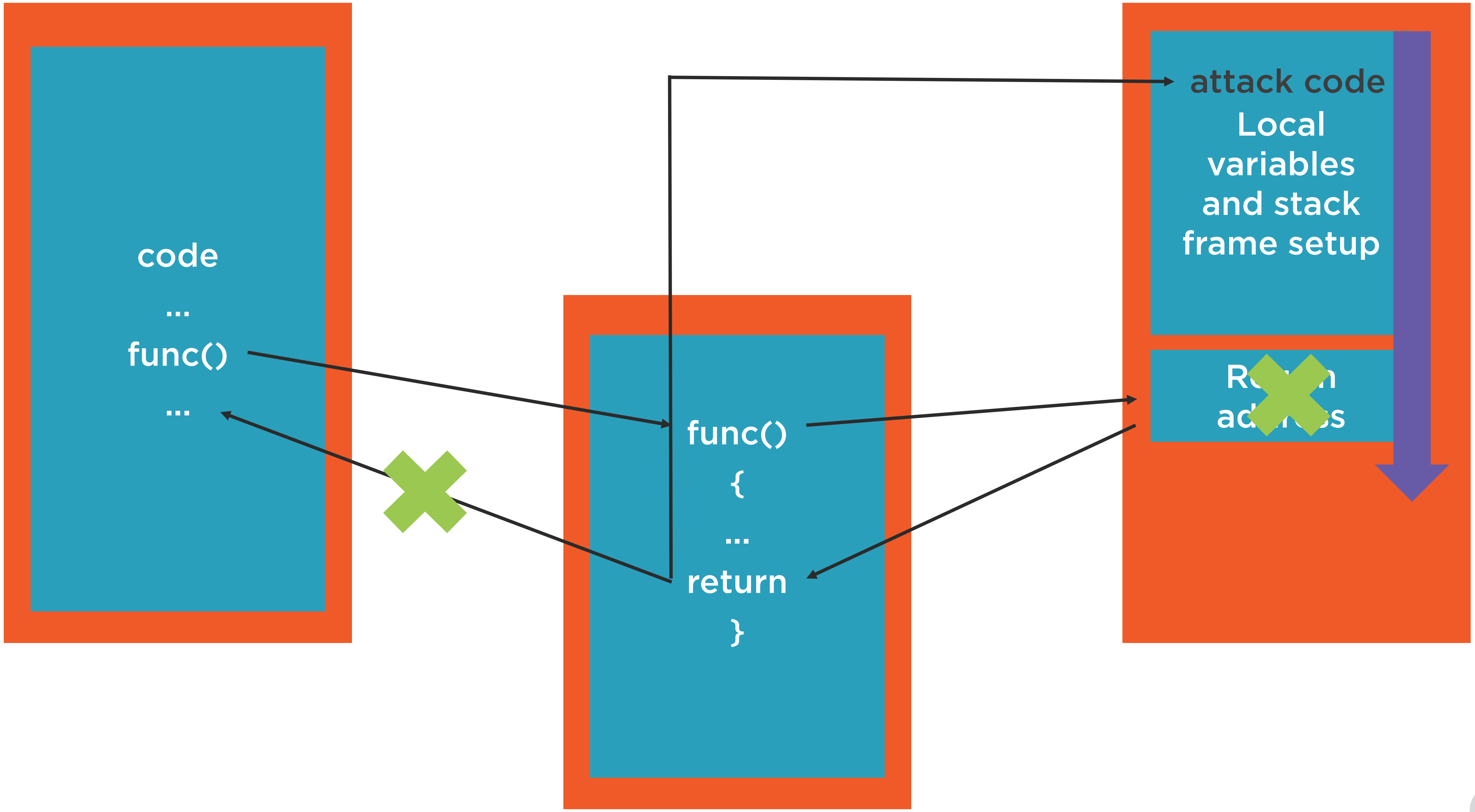
```c
int log_error(int farray, char *msg)
{
    char *err, *mesg;
    char buffer[24];

#ifdef DEBUG
    fprintf(stderr, "Mesg is at: 0x%08x\n", &mesg);
    fprintf(stderr, "Mesg is pointing at: 0x%08x\n", mesg);
#endif
    memset(buffer, 0x00, sizeof(buffer));
    sprintf(buffer, "Error: %s", mesg);

    fprintf(stdout, "%s\n", buffer);
    return 0;
}

int main(void)
{
    switch(do_auth())
    {
        case -1:
            log_error(ERR_CRITIC | ERR_AUTH, "Unable to login");
            break;
        default:
            break;
    }
    return 0;
}
```

```c
//sizeof(myObj) == 40

myObj *x = (myObj)malloc(sizeof(myObj));


memset(x, 0, si    (x));


memset(x, 0, sizeof(myObj));
```

Uninitialized memory

```
char * buf = malloc(1~~0~~0);
strncpy(buf, argv[1], strlen(argv[1]));
```

Heap buffer overflow


code
…
func()
…
func()
{
…
return
}
attack code
Local variables and stack frame setup
Return address

```
char buf[1024];
sprintf(buf, “%s@%s”, name, domain);
```

Unclear

```c
char buf[100];
for(int i=0; i<~~=~~100; i++)
    buf[i]=i;
```

Off-by-one

```
print[illegible]rgv[1]);
```

Format String

```
printf("%s\n", argv[1]);
```

```
         char buf[100];
unsigned int x = atoi(argv[1]);
         if ( x < 100 )
            strncpy(buf, argv[2], x);
```

Integer error

```
char buf[100];
int x = strlen(argv[2]);
if ( x < 100)
    strncpy(buf, argv[2], x);
```

Better, assuming argv[2] provided

```
switch(pkt->type){
    case 1:
        Auth(); break;
    case 2:
        Work(); break;
    case 3:
        Reset(); break;

}
fullyProcess(pkt);
```

No Default

It’s OK?

```
int main(int argc, char **argv) {
  char cat[] = "cat "; char *command; size_t commandLength;
  commandLength = strlen(cat) + strlen(argv[1]) + 1;
  command = (char *) malloc(commandLength);
  strncpy(command, cat, commandLength);
  strncat(command, argv[1], (commandLength - strlen(cat)) );
  system(command);
  return (0);
}
```

No, Command Injection

```
$ ./catWrapper Story.txt
When last we left our heroes...
```

```
$ ./catWrapper "Story.txt; ls"
When last we left our heroes...
Story.txt               doubFree.c              nullpointer.c
unstosig.c              www*                    a.out*
format.c                strlen.c                useFree*
catWrapper*             misnull.c               strlength.c
commandinjection.c      nodefault.c             trunc.c
```

```
w_char str[] = L”hello world!”

strlen(str);
```

Only returns 1

wcslen is a wide-character version of **strlen**

Ascii vs. Wide: Problem

```
void f( HINSTANCE hInst, UINT uID ) {
    TCHAR buff[128];
    if ( LoadString ( hInst, uID, buff, sizeof(buff) ) )
    {
        // code...
    }
}
```

sizeof returns number of bytes, which is twice the elements we need

Ascii vs. Wide: Fix

```
#define _countof(array) (sizeof(array)/sizeof(array[0]))

void f( HINSTANCE hInst, UINT uID ) {
    TCHAR buff[128];
    if ( LoadString ( hInst, uID, buff, _countof(buff) ) )
    {
        // code...
    }
}
```

Use-after-free or double free?

```
char *foo(char *ptr, char len){
  char *tmp;
  tmp = realloc(ptr, len);
  if (!tmp)
    return tmp;
  ptr = tmp;
  return ptr;
}
```

if len is 0, realloc acts like free!

calling function needs to validate return, don’t use or free again!

Review memory allocations closely

- Validate input (size)
  - Prefer hard limits when possible
- Watch for math in allocation
- Copy needs same math
- Wild read or writes can be a problem

Bad

No limit

```
buf5 = malloc(strlen(argv[8]) + strlen(argv[9]) + 2);
strcpy(buf5, argv[8]);
strcat(buf5, argv[9]);
```

Different math

Fix

```
if(!argv[8] || !argv[9])
   return bad_inputs;
size_1 = strlen(argv[8]);  size_2 = strlen(argv[9]);
c_size = size_1 + size_2 + 2;
if( c_size < size_1 || c_size < size_2 || c_size > limit )
   return size_error;
buf5 = malloc(c_size);
if(!buf5)
   return alloc_error;
strncpy(buf5, argv[8]);  strncat(buf5, argv[9]);
```

Review Allocations

```
KpUInt32_t Index, Limit;    KpUInt16_t FAR *UInt16Ptr;
jlong bufSizeL;    jint bufSize;

   Limit = SpGetUInt32 (Buf);

   if (0 == Limit) {

      Curve->Count = Limit;

      Curve->Data = NULL;

      return SpStatSuccess;

   }
```

Some validation of input… that’s good

Review Allocations

```
bufSizeL = (jlong)Limit * sizeof(*UInt16Ptr);
bufSize = (jint)bufSizeL;


if (bufSizeL != bufSize)
    return SpStatBadProfile;
```

Clever validation...

Could use hard limit also

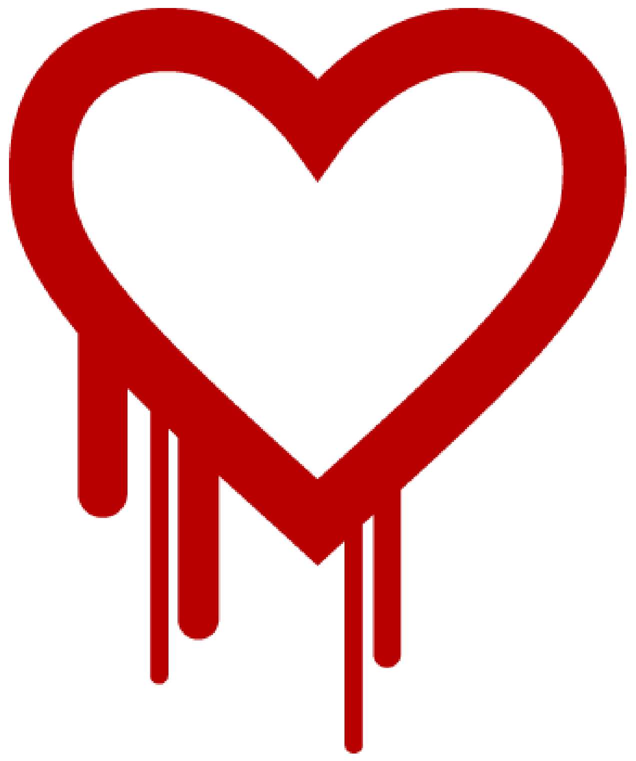
Review Allocations

```
UInt16Ptr = (KpUInt16_t *)SpMalloc (bufSize);
if (NULL == UInt16Ptr)
    return SpStatMemory;
Curve->Count = Limit;
Curve->Data = UInt16Ptr;
for (Index = 0; Index < Limit; Index++)
    *UInt16Ptr++ = SpGetUInt16 (Buf);
return SpStatSuccess;
```

The allocation; good to check ptr

Copy looks OK

Could we read out of bounds?



Out of bounds write is clearly bad

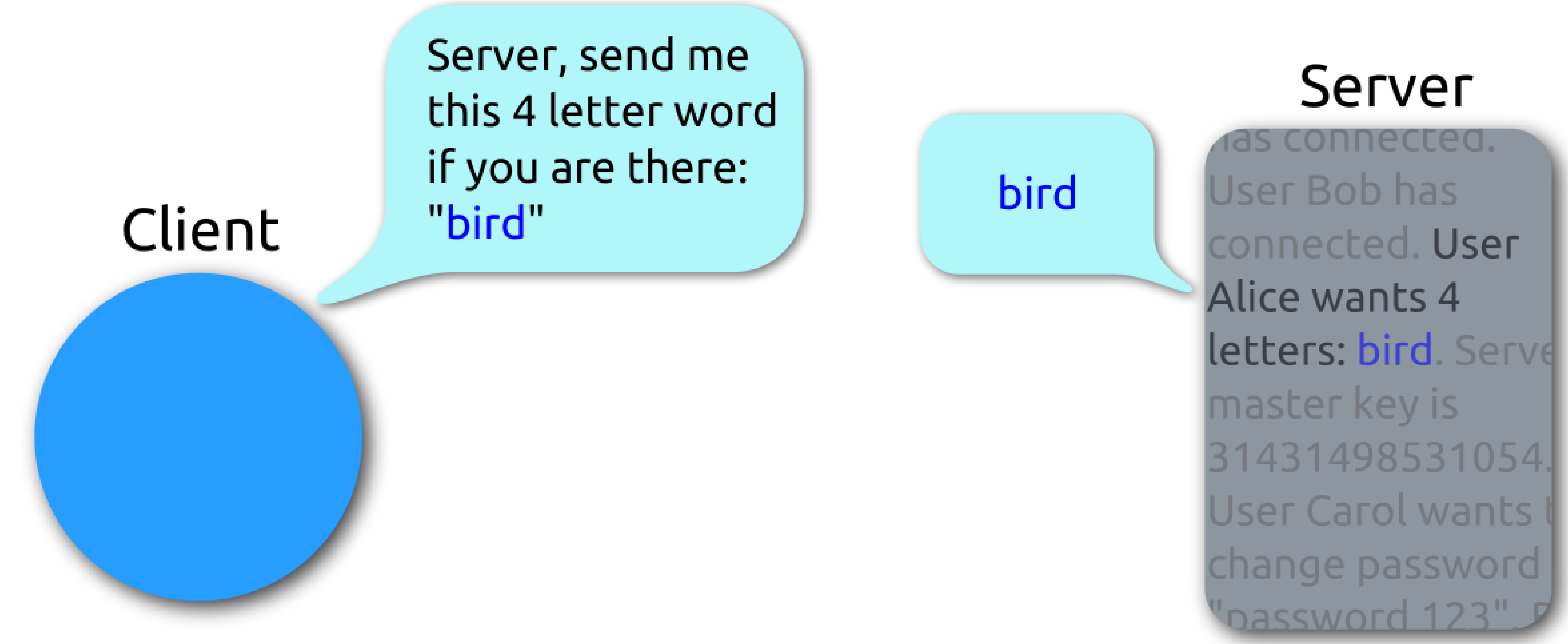
- But read?
  - Heartbleed is a security bug disclosed in April 2014 in the OpenSSL cryptography library
  - Improper input validation in TLS heartbeat extension
    - Buffer over-read

Demo

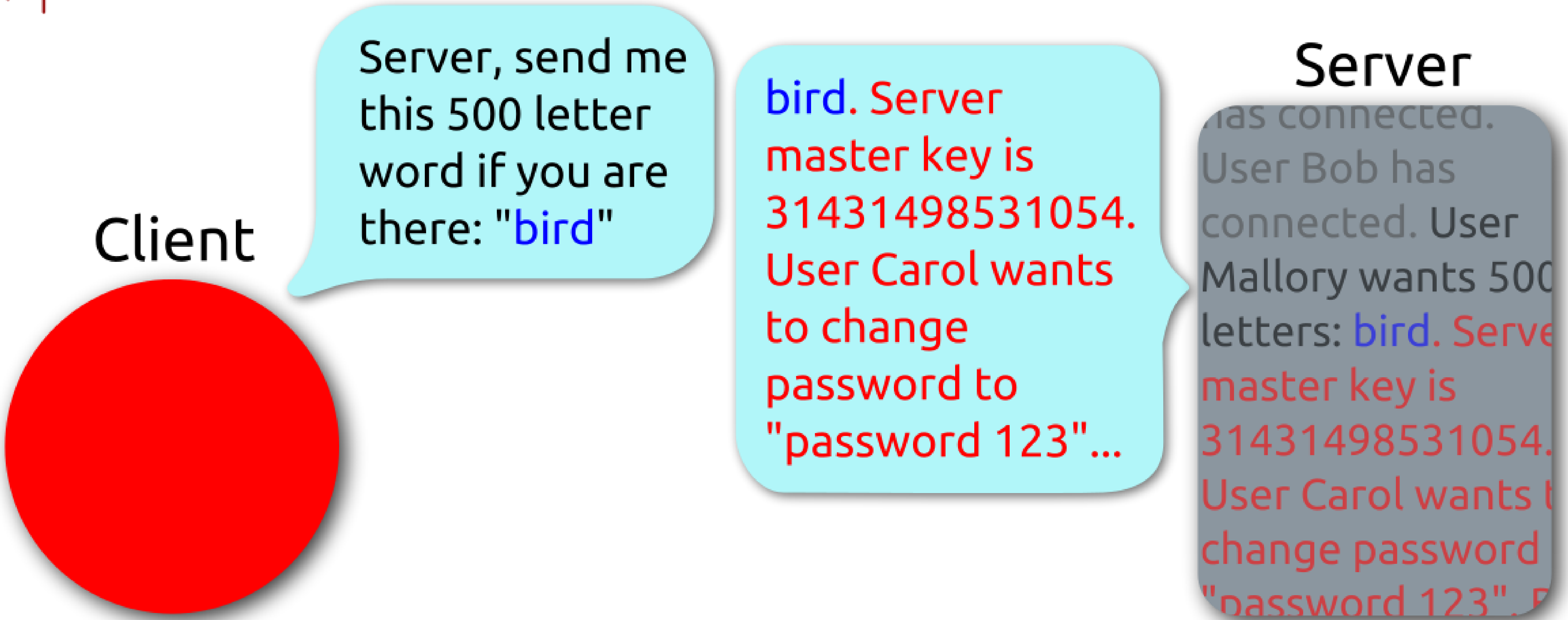
View Heartbleed

- Show SCI Understand
  - Code navigation
  - Differencing

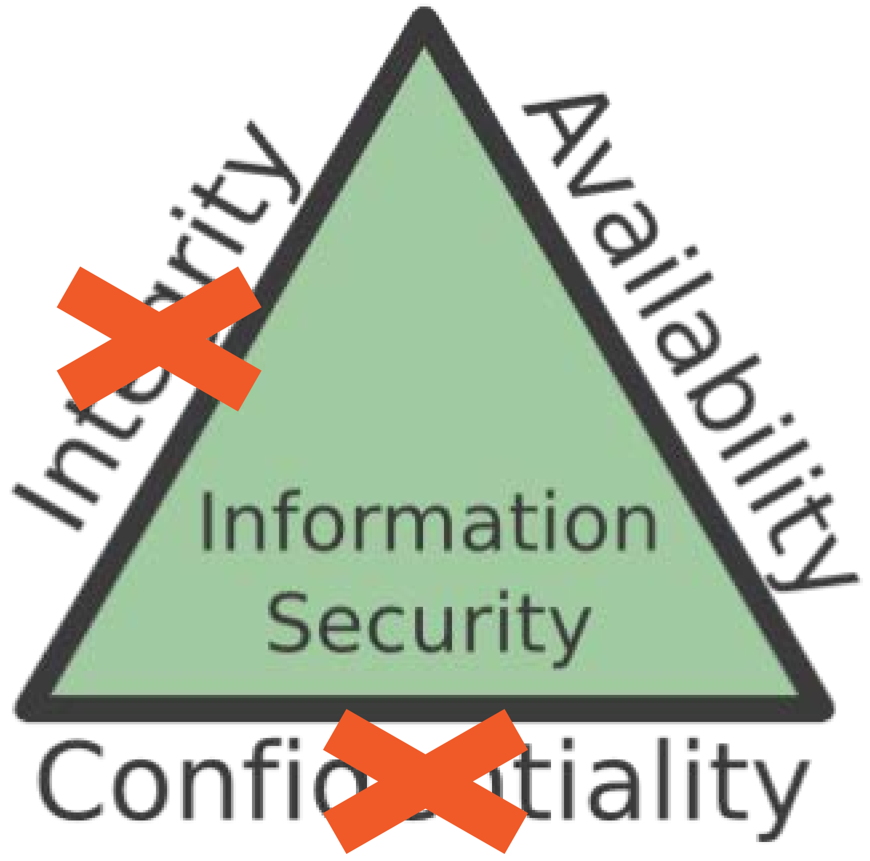
Heartbeat – Normal usage


Server, send me this 4 letter word if you are there: "bird"
bird
Server
Client
as connected. User Bob has connected. User Alice wants 4 letters: bird. Serve master key is 31431498531054. User Carol wants t change password "password 123". P

Heartbeat – Malicious usage


Client
Server, send me this 500 letter word if you are there: "bird"
bird. Server master key is 31431498531054. User Carol wants to change password to "password 123"...
Server
as connected. User Bob has connected. User Mallory wants 500 letters: bird. Serve master key is 31431498531054. User Carol wants t change password "password 123".

CIA


Integrity
Availability
Information Security
Confidentiality

Heartbleed Post-mortem

- Submitted by student (OpenSSL)
  - Reviewed and accepted
  - Existed for about 3 years
    - Unknown to most
      - Found by Neel Mehta and Codenomicon
- Blame project or users for not investing properly?
  - Under staffed and under reviewed?
  - ~500,000 lines of critical code
  - Use of macros made static analysis tools fail

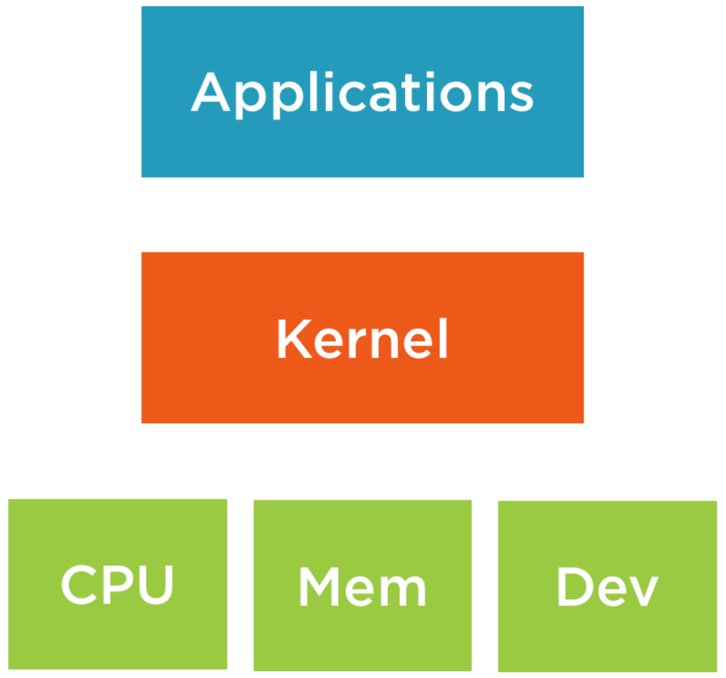
Heartbleed Post-mortem

- Lack of
  - Proper design
    - Overly complex structures
      - E.g. Wrote their own memory management structures
  - Testing
    - Negative tests
  - Threat modeling
    - Risk analysis before coding
  - Penetration testing
    - Security code audit
    - Fuzzing

Heartbleed
Post-mortem

Summary

- 17% (around half a million) of Internet's secure web servers were believed vulnerable
- Allowing theft of the servers' private keys and users' session cookies and passwords
- Huge negative impact + cost



Kernels have vulnerabilities too?

- Of course

System code in multiuser system

- Need a copy of data from apps
- Else, double fetch can happen

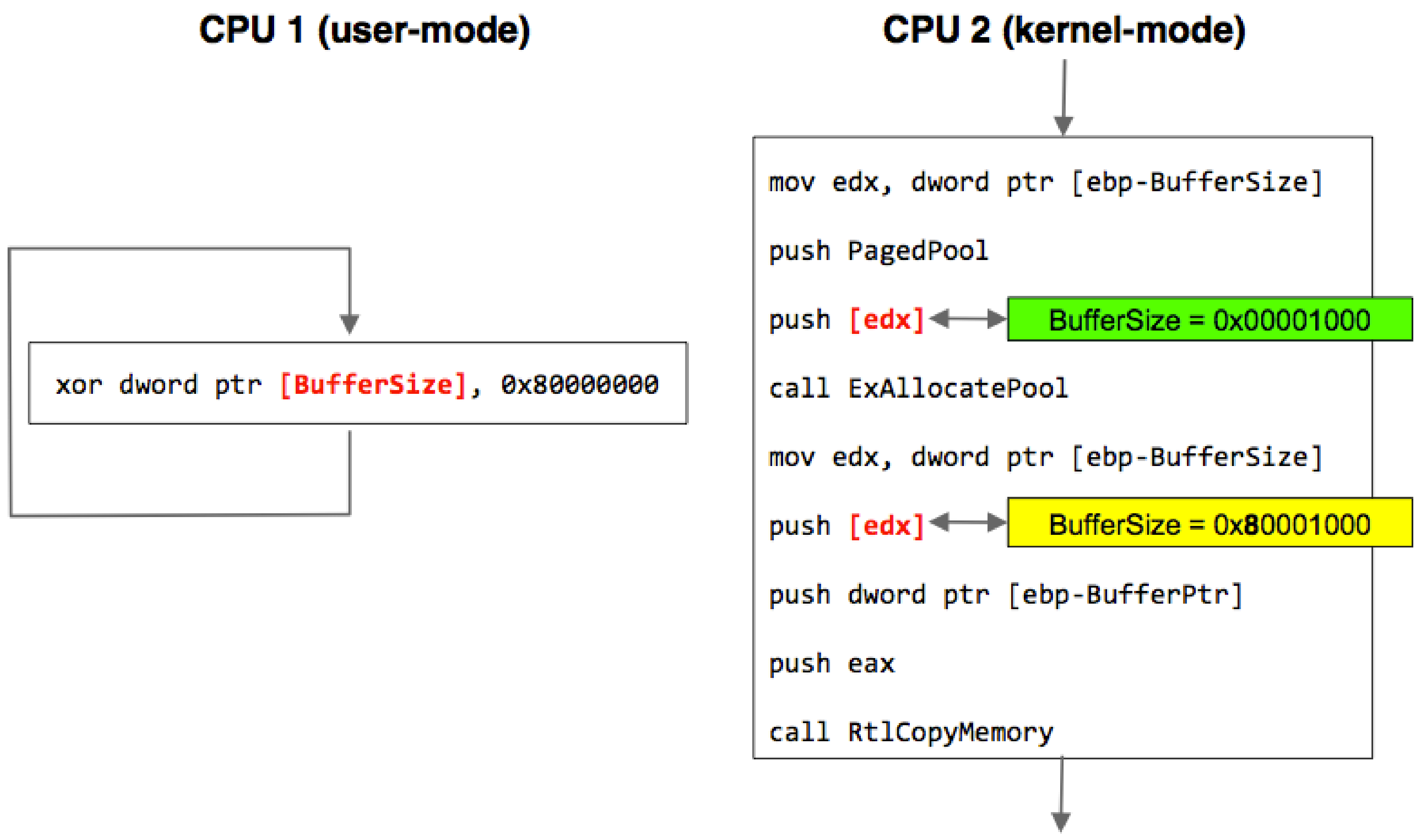
This is OK?

```
PDWORD BufferSize = /* controlled user-mode address */;
PBYTE BufferPtr = /* controlled user-mode address */;
PBYTE LocalBuffer;

LocalBuffer = ExAllocatePool(PagedPool, *BufferSize);
if (LocalBuffer != NULL) {
  RtlCopyMemory(LocalBuffer, BufferPtr, *BufferSize);
} else {
  // bail out
}
```

Time-of-check

Time-of-use

Double Fetch



```
xor dword ptr [BufferSize], 0x80000000
```

```
mov edx, dword ptr [ebp-BufferSize]
push PagedPool
push [edx]
call ExAllocatePool
mov edx, dword ptr [ebp-BufferSize]
push [edx]
push dword ptr [ebp-BufferPtr]
push eax
call RtlCopyMemory
```

Fetch twice - bad

```
__try {
  ProbeForWrite(*UserPtr, sizeof(STRUCTURE), 1);
  (*UserPtr)->Field = 0;
} except {
  return GetExceptionCode();
}
```

vs.

Fetch once - good

```
PSTRUCTURE Pointer;
__try {
  Pointer = *UserPtr;

  ProbeForWrite(Pointer, sizeof(STRUCTURE), 1);
  Pointer->Field = 0;
} except {
  return GetExceptionCode();
}
```

Compliers Could Remove Code?

- If undefined
  - Compiler can remove per spec
    - Example from MIT paper
    - http://pdos.csail.mit.edu/~xi/papers/stack-sosp13.pdf
  - Patched now of course

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return;   /* len too large */
if (buf + len < buf)
    return;   /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

The c standard states that an overflowed pointer is undefined

A pointer overflow check found in several code bases. The code becomes vulnerable as gcc optimizes away the second if statement.

Audit program

- Overflows
- Command injection
- Format string
- Macro issue
  - Hint

Summary

Common mistakes in C

- Over write/read
- Integer
- Format string
- Logic
- Uninitialized memory
- Use-after-Free
- Double fetch
- Macros
- Compiler errors