



Murdoch
UNIVERSITY

Software Security

ICT287 Computer Security



Admin

- After this lecture start your research into vulnerabilities for the project

Security Flaws

- In general security flaw appears where an attacker tries something that developer didn't consider adequately
- Maybe attacker accesses service that developer thought was hidden
- OR maybe two components interact in a way that is insecure

Flaws can typically categorised as being either **design flaws** or **implementation flaws**

Design Flaws

- Flaws in design of software or system, i.e. design does not meet actual requirements
 - Design errors in crypto algorithms or cryptographic PRNGs
 - Missing or inappropriate security mechanisms
 - Actual requirements changed from those assumed during design (common issue!)
- Its hard to predict just how a system will be used, so design flaws can be hard to prevent
- Can be hard to deal with (need redesign)
- But following some best practices and mitigations goes a long way

Implementation Flaws

- Flaws in implementation of software or system, i.e. software does not meet design (bugs)
 - Buffer overflows/overreads due to missing/incorrect length checking
 - Inappropriate checking/sanitisation of untrusted user input
 - Wrong/inappropriate runtime configuration
- Implementation flaws, while often obscure in their specifics, tend to be simpler to deal with
- Just fix implementation

The Attacker's Objective...

- Often classic *remote code execution, privilege escalation*
- In other words: attacker wants to do something that he should not be allowed to do!
- Gain additional access... To read others files, or change how a system works
- Or maybe just to crash system



Impact of Flaw/Vulnerability

- We may classify flaws based on several criteria
- After all, its sometimes necessary to prioritise which ones we address first

Common Vulnerabilities and Exposures (CVE)

- Common Vulnerabilities and Exposures (CVE) is dictionary of common names (i.e. CVE Identifiers) for publicly known information security vulnerabilities
- CVE's common identifiers make it easier to share data across separate network security databases and tools, and provide baseline for evaluating coverage of organization's security tools
- If report from one of your security tools incorporates CVE Identifiers, you may then quickly and accurately access fix information in one or more separate CVE-compatible databases to remediate problem

CVE

- ☐ Standardized names for vulnerabilities
- ☐ Standard descriptions
- ☐ Way to interoperate between security tools
- ☐ Baseline for evaluation of coverage of tools
- ☐ Free to use
- ☐ Industry endorsed

About 20 numbering authorities

[National Cybersecurity FFRDC](#), operated by the [Mitre Corporation](#), maintains the system, with funding from the [National Cyber Security Division](#) of the [United States Department of Homeland Security](#) ([Wikipedia](#))

MODIFIED

Identifier

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Current Description

The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.

Source: MITRE

[+View Analysis Description](#)

Description

Severity

CVSS Version 3.x

CVSS Version 2.0

CVSS 3.x Severity and Metrics:



NIST: NVD

Base Score: 7.5 HIGH

Metric Details

Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CVSS v3.1 Severity and Metrics:

Base Score: 7.5 HIGH

Vector: AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

Impact Score: 3.6

Exploitability Score: 3.9

Attack Vector (AV): Network

Attack Complexity (AC): Low

Privileges Required (PR): None

User Interaction (UI): None

Scope (S): Unchanged

Confidentiality (C): High

Integrity (I): None

Availability (A): None

Evaluator Impact

Severity score

CVSS V2 scoring evaluates the impact of the vulnerability on the host where the vulnerability to your organization, take into account the nature of the data that is being risk acceptance. While CVE-2014-0160 does not allow unrestricted access to memory information from memory locations which have the potential to contain particularly sensitive passwords. Theft of this information could enable other attacks on the information system sensitivity of the data and functions of that system.

References to Advisories, Solutions, and Tools

By selecting these links, you will be leaving NIST webpage. We have provided these links for your information.

Common Weakness Enumeration (CWE)

- Common Weakness Enumeration (CWE) provides unified, measurable set of software weaknesses
- Enables discussion, description, selection, and use of software security tools to find these weaknesses in source code and operational systems
- Also gives better understanding and management of software weaknesses related to architecture and design

Common Weakness Enumeration (CWE)

CWE VIEW: Research Concepts

View ID: 1000
Type: Graph

Downloads: [Booklet](#) | [CSV](#) | [XML](#)

Objective

This view is intended to facilitate research into weaknesses, including their inter-dependencies, and can be leveraged to systematically identify theoretical gaps within CWE. It is mainly organized according to abstractions of behaviors instead of how they can be detected, where they appear in code, or when they are introduced in the development life cycle. By design, this view is expected to include every weakness within CWE.

Audience

Stakeholder	Description
Academic Researchers	Academic researchers can use the high-level classes that lack a significant number of children to identify potential areas for future research.
Vulnerability Analysts	Those who perform vulnerability discovery/analysis use this view to identify related weaknesses that might be leveraged by following relationships between higher-level classes and bases.
Assessment Tool Vendors	Assessment vendors often use this view to help identify additional weaknesses that a tool may be able to detect as the relationships are more aligned with a tool's technical capabilities.

Relationships

The following graph shows the tree-like relationships between weaknesses that exist at different levels of abstraction. At the highest level, categories and pillars exist to group weaknesses. Categories (which are not technically weaknesses) are special CWE entries used to group weaknesses that share a common characteristic. Pillars are weaknesses that are described in the most abstract fashion. Below these top-level entries are weaknesses are varying levels of abstraction. Classes are still very abstract, typically independent of any specific language or technology. Base level weaknesses are used to present a more specific type of weakness. A variant is a weakness that is described at a very low level of detail, typically limited to a specific language or technology. A chain is a set of weaknesses that must be reachable consecutively in order to produce an exploitable vulnerability. While a composite is a set of weaknesses that must all be present simultaneously in order to produce an exploitable vulnerability.

Show Details: ☐

[Expand All](#) | [Collapse All](#) | [Filter View](#)

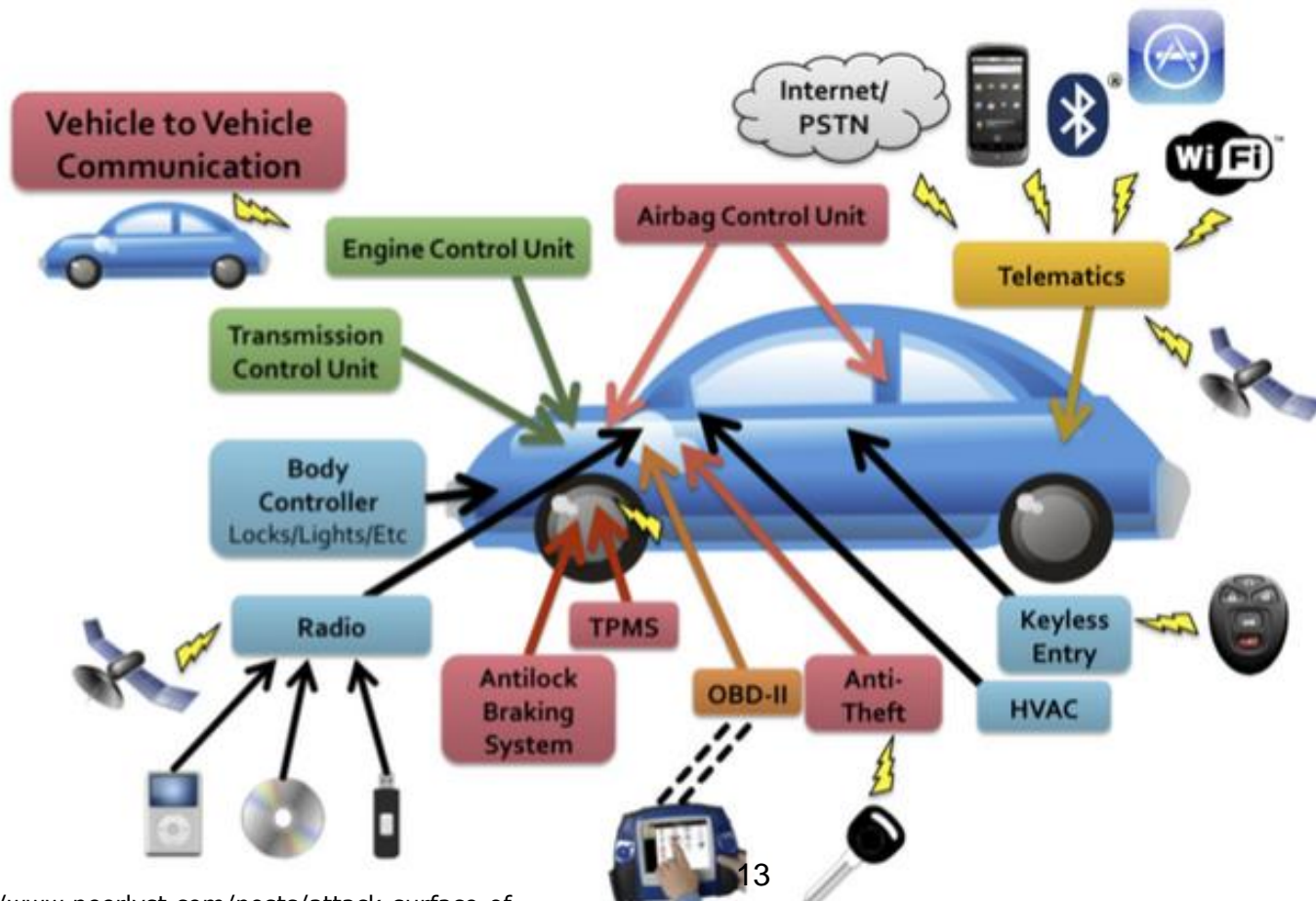
1000 - Research Concepts

- [-] [P] Improper Access Control - (284)
- [-] [P] Improper Interaction Between Multiple Correctly-Behaving Entities - (435)
- [-] [P] Improper Control of a Resource Through its Lifetime - (664)
- [-] [P] Incorrect Calculation - (682)
- [-] [P] Insufficient Control Flow Management - (691)
- [-] [P] Protection Mechanism Failure - (693)
- [-] [P] Incorrect Comparison - (697)
- [-] [P] Improper Check or Handling of Exceptional Conditions - (703)
- [-] [P] Improper Neutralization - (707)
- [-] [P] Improper Adherence to Coding Standards - (710)

[BACK TO TOP](#)

Attack Surface

System attack surface: possible entry points for attacker



Attack Surface

When performing high-level design, one will already have defined components with which attacker can interact with, giving highest-level notion of entry points

Define mechanisms through which anyone could interact with application

- Open network ports
- File IO
- Local UI elements
- IPC
- Public methods

Document each

- Unambiguous description
- Unique ID
- Document program entry points as they are identified
- Increasing granularity as project proceeds

Attack Surface Changes

Attack surface of many systems is fluid

What sort of things change attack surface of an application?

- Adding something to infrastructure
- Software changes (e.g. updates/patches)
- Changes in organisation
- ...

CWE/SANS Top 25

- Top 25 Most Dangerous Software Errors is list of most widespread and critical errors that can lead to serious vulnerabilities in software
- They are often easy to find and easy to exploit
- They are dangerous because they will frequently allow attackers to completely take over the software, steal data or prevent software from working at all

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

CWE/SANS Top 25

- Many similarities in list items: good news as one mitigation may protect against several items
- By the end of this unit you will understand all of items on this list
- In second part of lecture we will discuss infamous buffer overflow vulnerabilities and several related vulnerabilities

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲















What is Buffer Overflow?

- Bug that affects low level code (often C/C++), with significant implications
- Often program will just crash
- But attacker can use this to do a lot more
 - **Run code of attacker's choice**
 - **Steal or corrupt data**

Buffer Overflows

Not only are they still common, but they share common features with many other bugs

C and C++ are still hugely popular languages
(Source IEEE Programming Languages Survey 2014)

Language Rank	Types	Spectrum Ranking
1. Java	  	100.0
2. C	  	99.2
3. C++	  	95.5
4. Python	 	93.4
5. C#	  	92.2

Critical Systems

What languages do you think are used for systems such as:

OS Kernels

Servers

Embedded systems

Some history! Morris Worm

- Robert Morris, then Cornell grad student released this worm in 1988
- It exploited buffer overflow attack on VAX systems to propagate and was intended to propagate slowly and measure Internet

<http://weburg.net/news/6955>

However, overflow attack also caused Sun based systems to crash instead of harmlessly replicating. Worm also replicated so fast that even VAX systems were brought down.

Morris was convicted and sentenced to 3 years probation. He is now a professor at MIT.



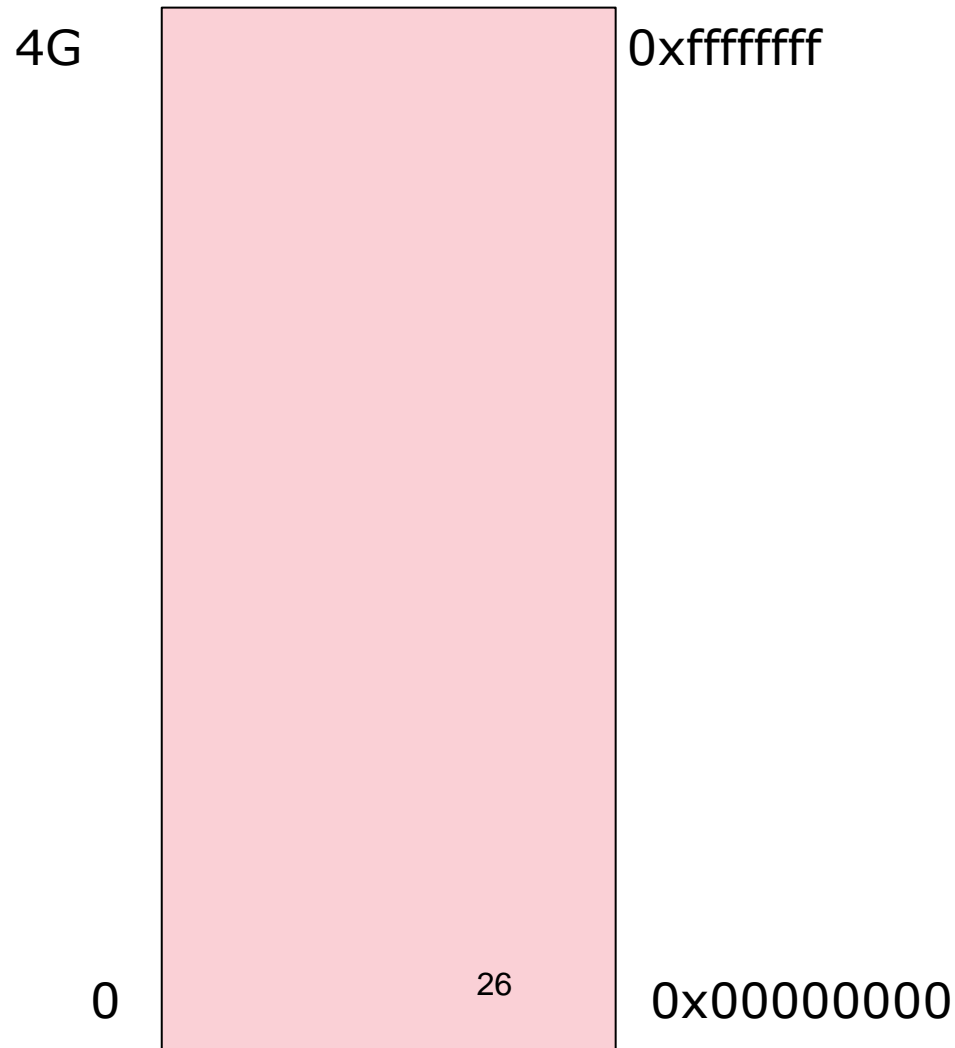
More Recently

- 2001: CodeRed exploited overflow in MS IIS and infected 300,000 machines in 14 hours
- 2003: SQL Slammer exploited overflow in MS-SQL and infected 75,000 machines in 10 minutes
- 2019: buffer overflow in WhatsApp allowed device takeover, buffer overflow in Exim (powers more than half of the world's mail servers) allowed DoS or even remote code execution
- 2021: buffer overflow in sudo can be used for privilege escalation to root

Memory Layout

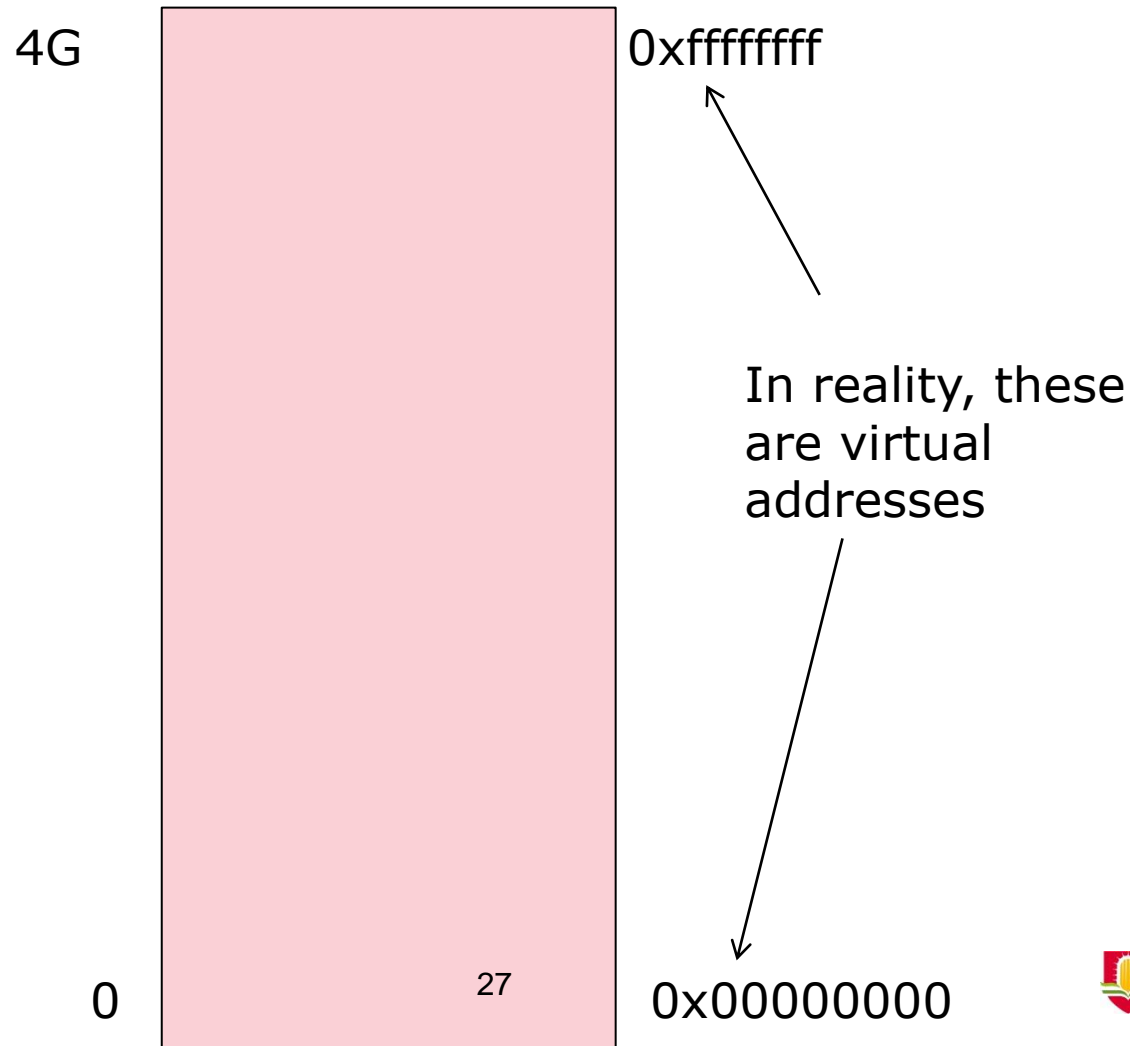
- This is revision from your first year systems unit
- How is program data laid out in memory?
- What does stack look like?
- What effect do function calls have on memory?
- We will talk about 32bit first as it is easier to understand

Memory Layout

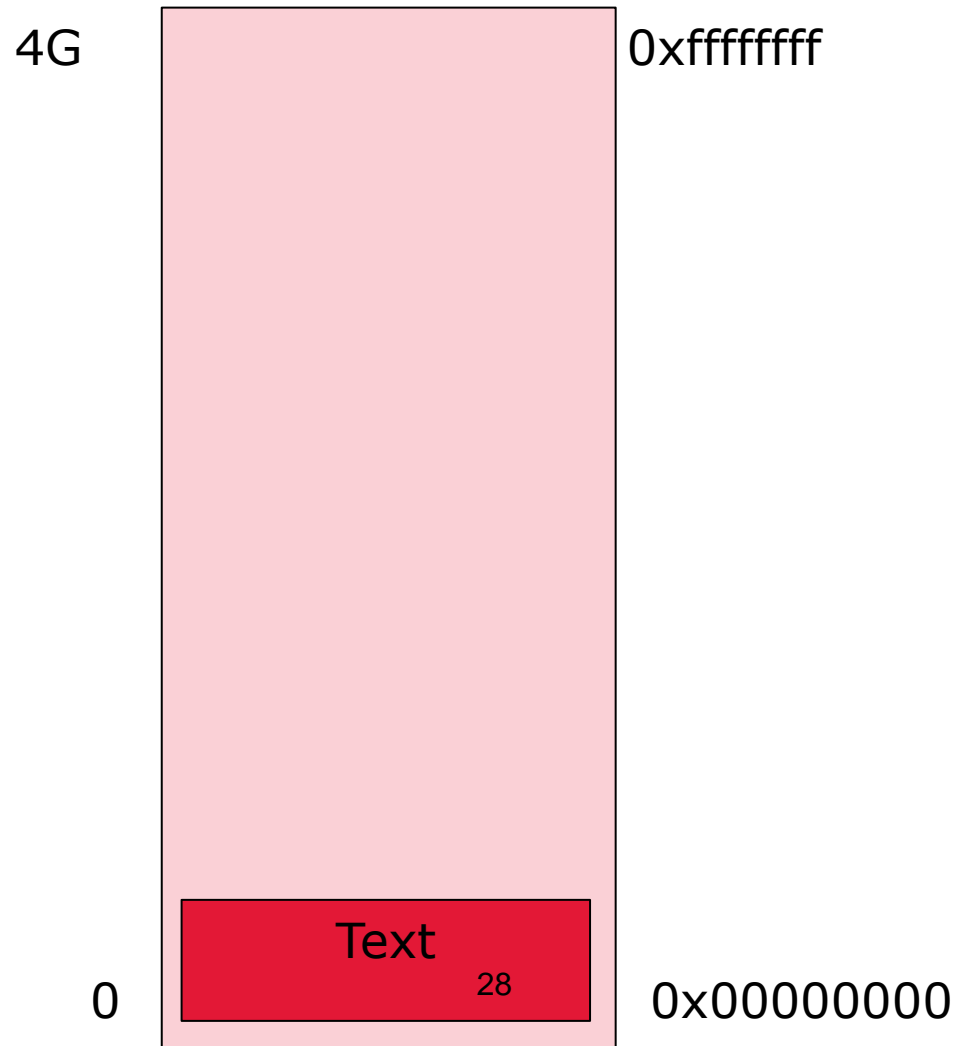


Memory Layout

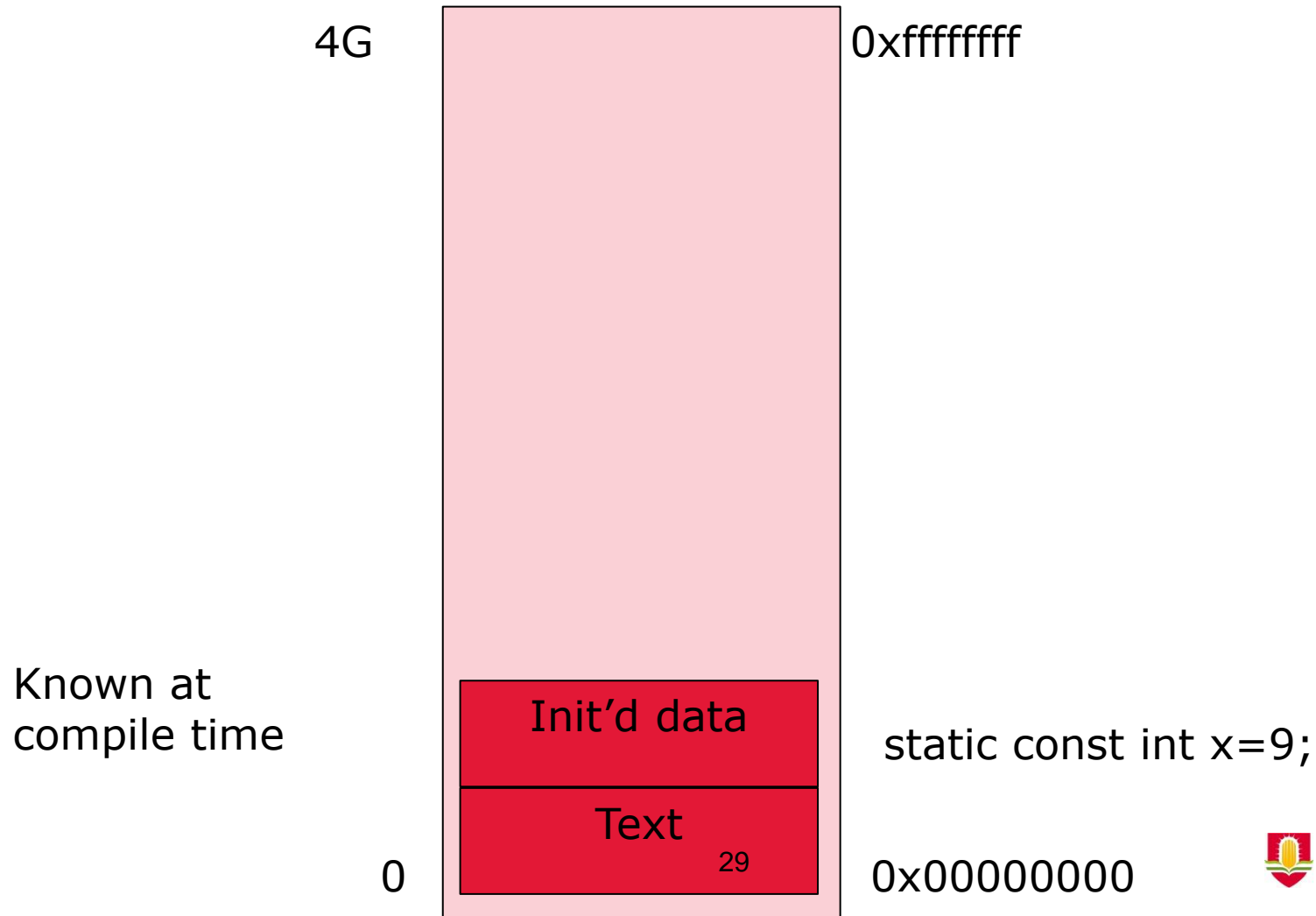
The processes view is that it owns all of it



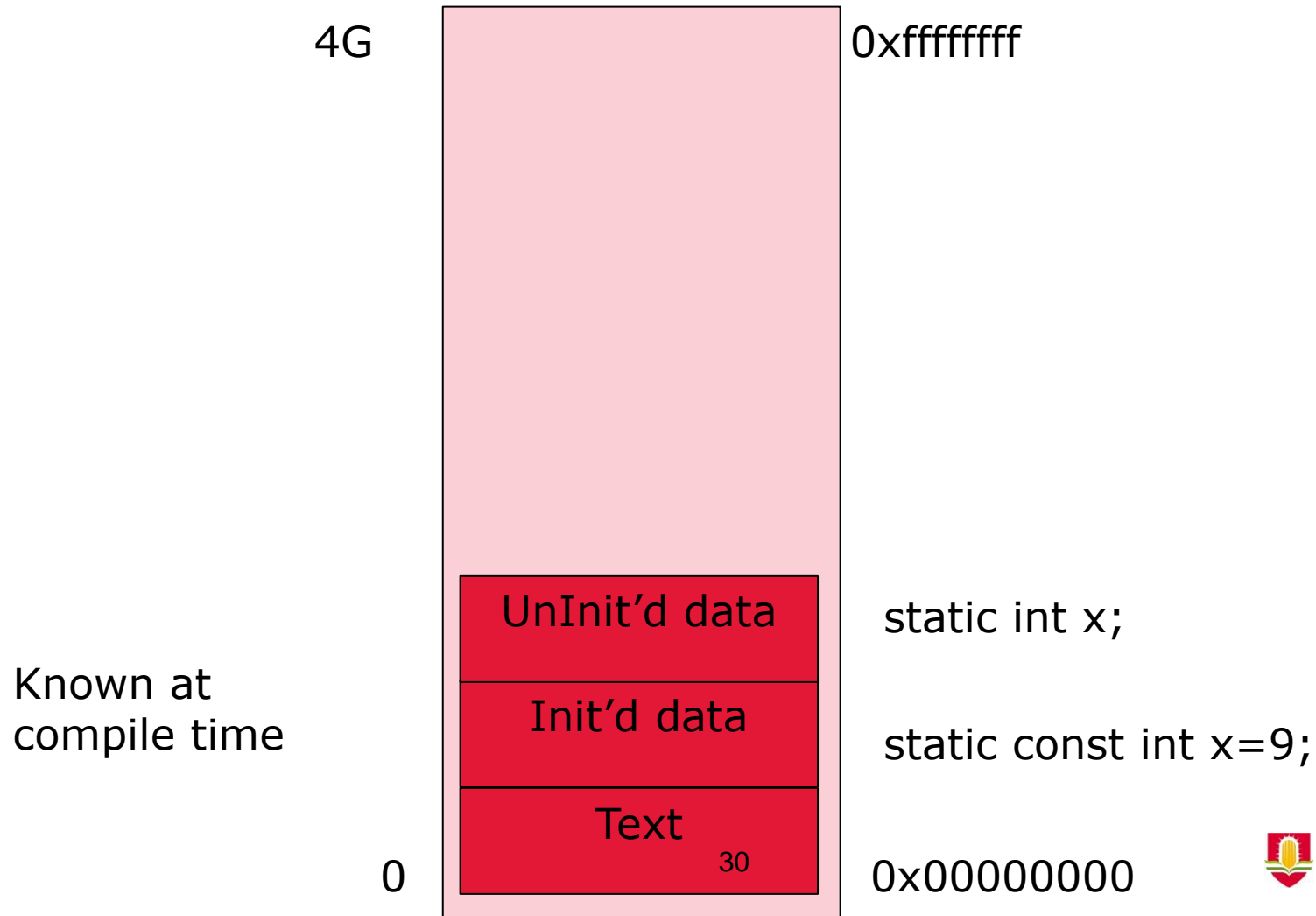
Location of Data Areas



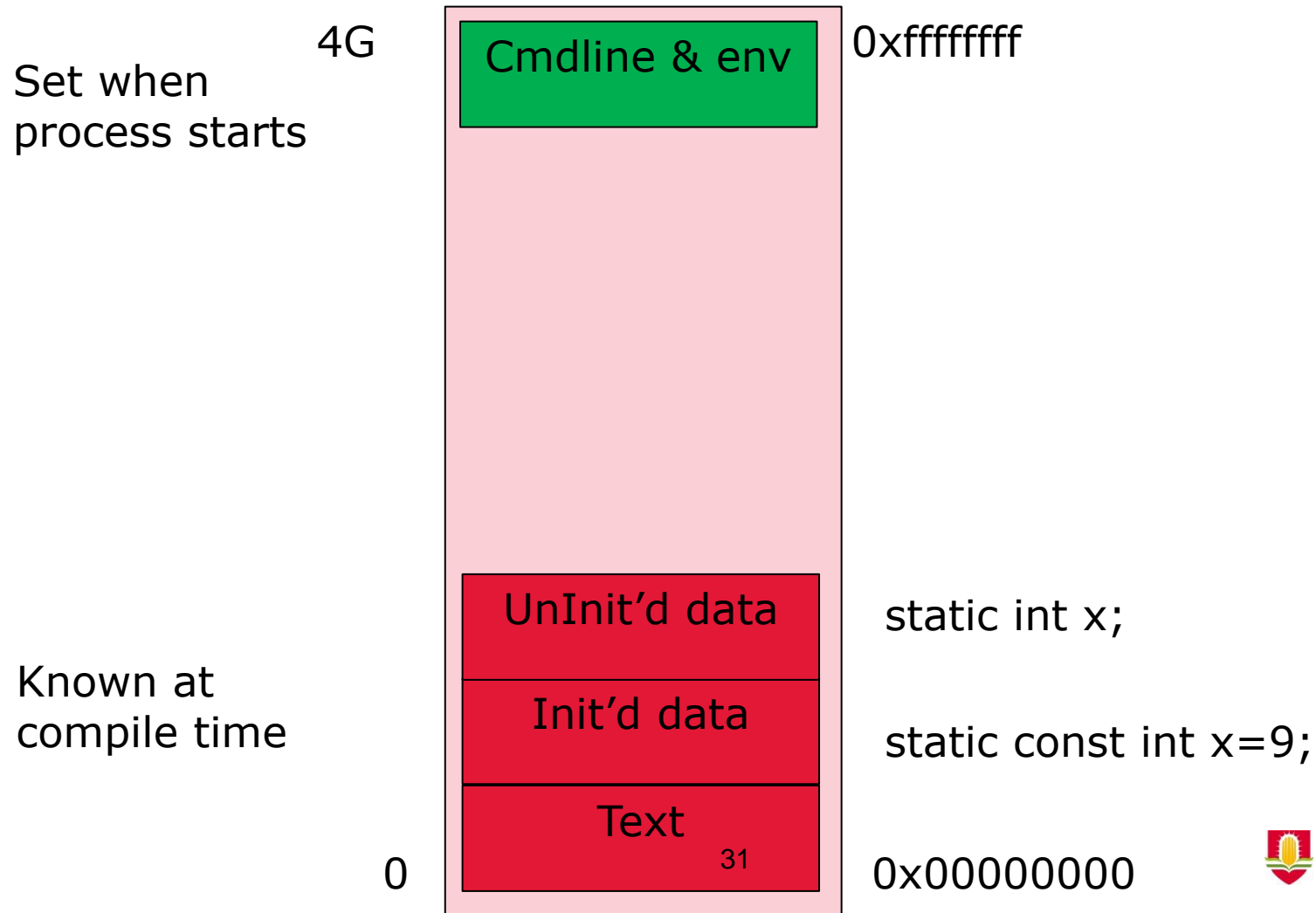
Location of Data Areas



Location of Data Areas



Location of Data Areas



Location of Data Areas

Set when
process starts

Cmdline & env

0xffffffff

Stack

int f() {int x;...}

Runtime

Heap

malloc(64*sizeof(long));

UnInit'd data

static int x;

Init'd data

static const int x=9;

Known at
compile time

Text

32

0x00000000

Memory Allocation

Heap grows upwards and stack grows downwards



Memory Allocation

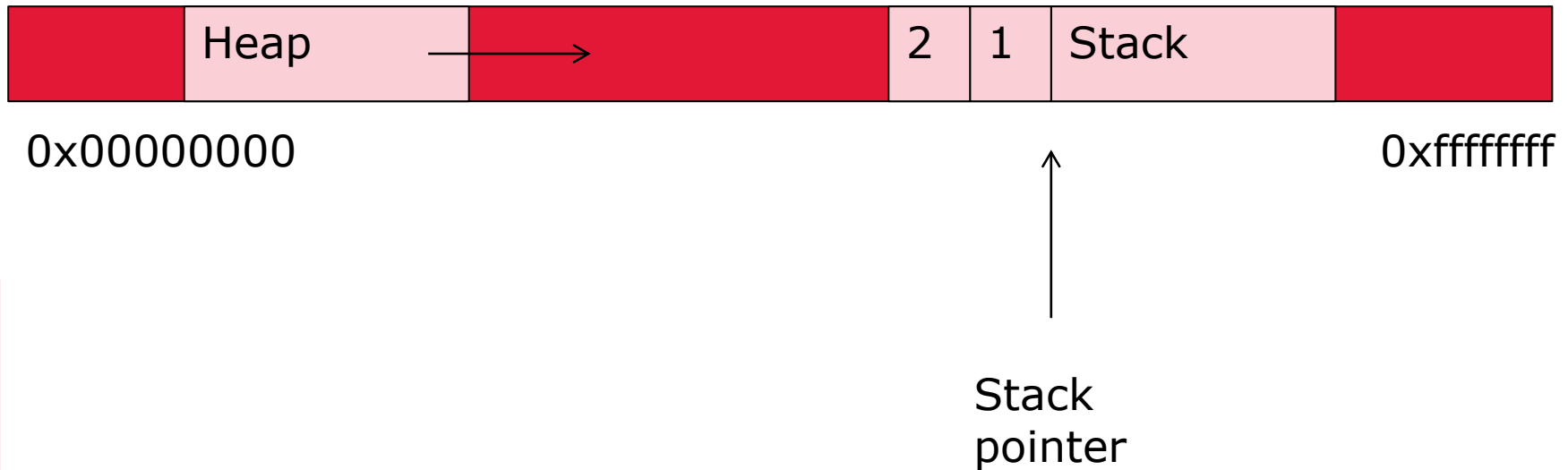
Stack pointer keep track of end of stack



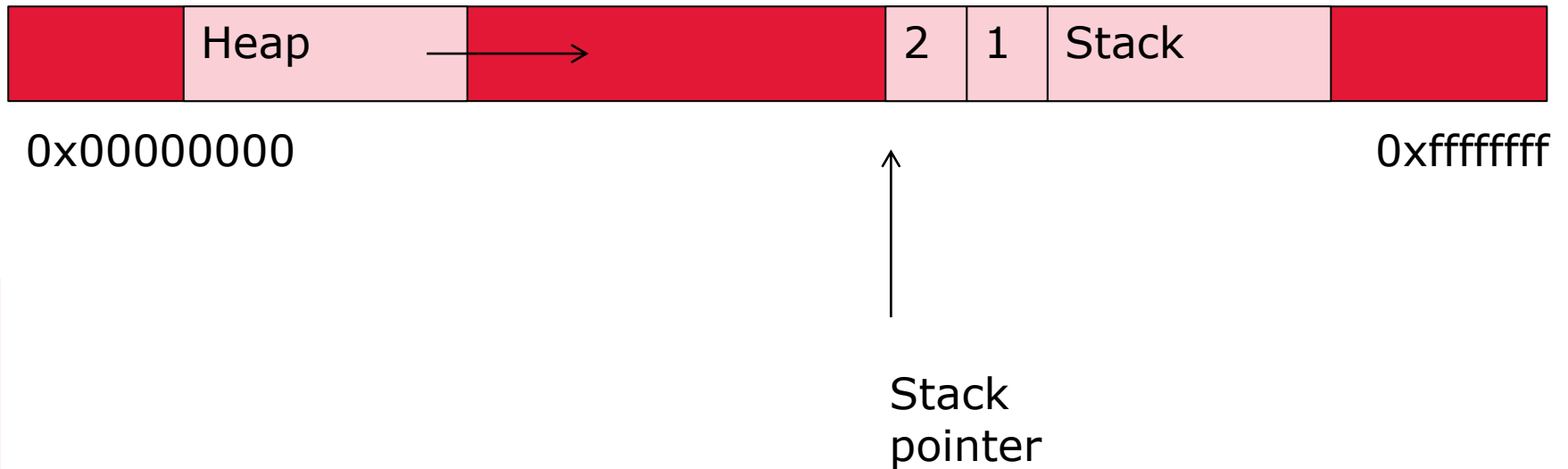
Memory Allocation

Variables can be pushed on Stack

push 1
push 2



Memory Allocation



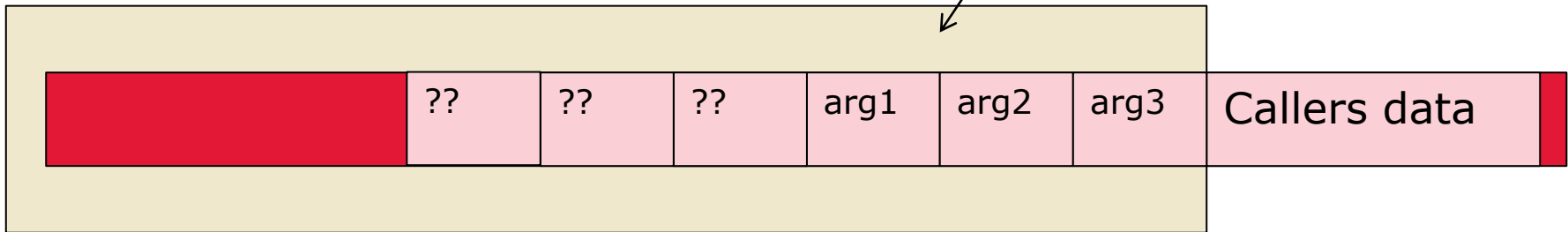
Stack and Function Calls

- What happens when we call function?
- What happens when we return from function?

Calling a Function

```
void func (char *arg1, int arg2, int arg3)
{
...
}
```

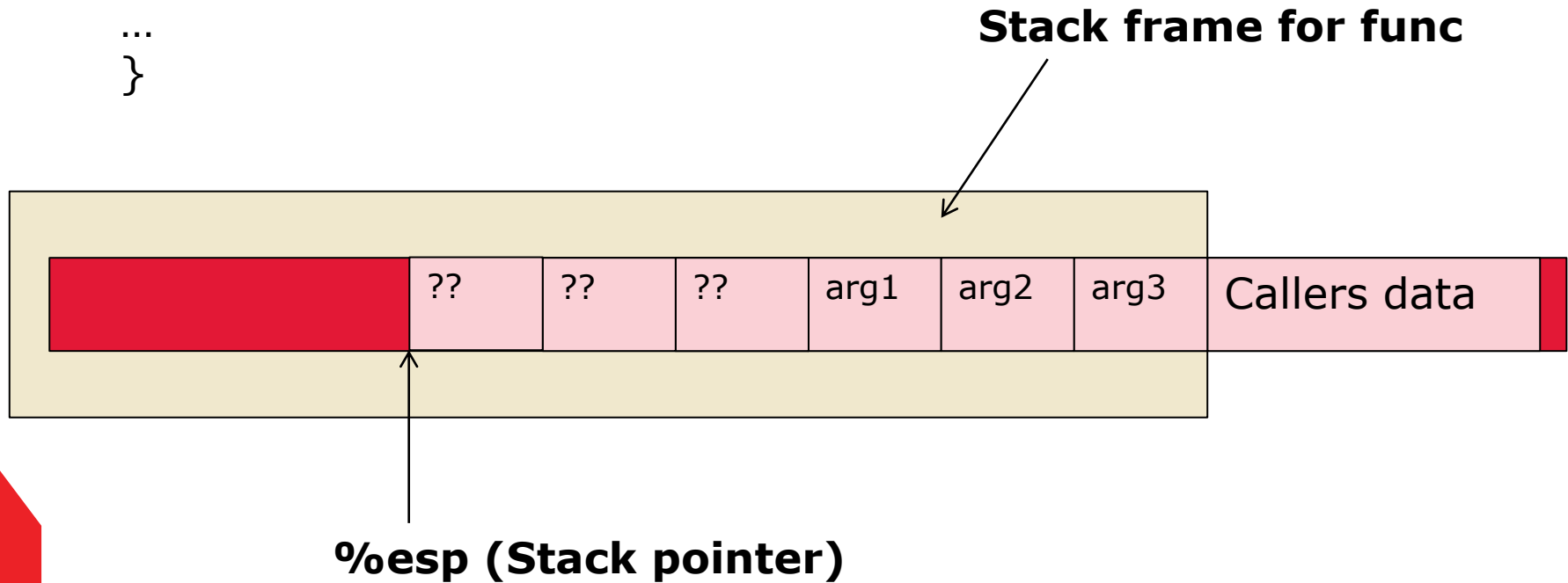
Stack frame for func



Arguments pushed in
reverse order of code

Calling a Function

```
void func (char *arg1, int arg2, int arg3)
{
...
}
```

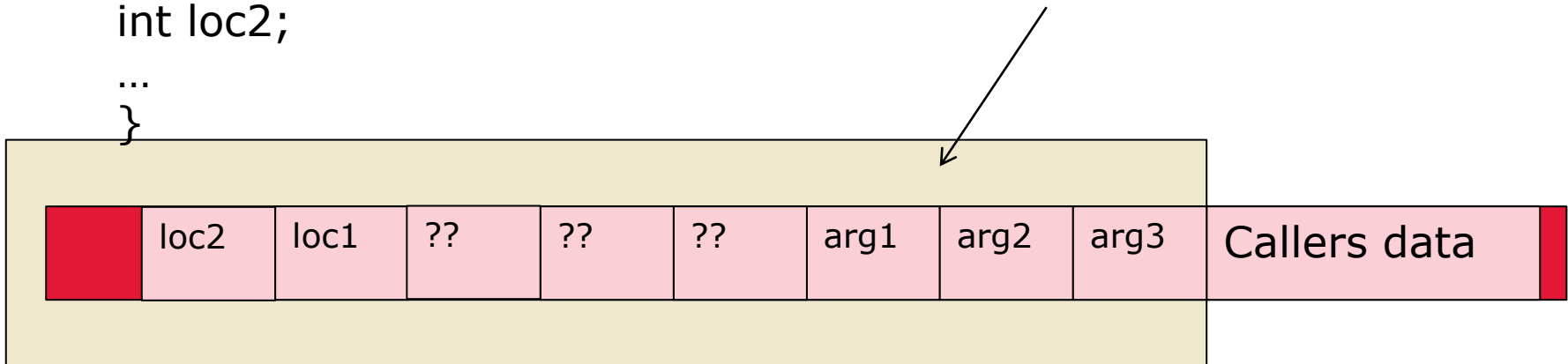


- Stack pointer always points to top of stack

Calling a Function

```
void func (char *arg1, int arg2, int arg3)
{
  char loc1[4];
  int loc2;
  ...
}
```

Stack frame for func

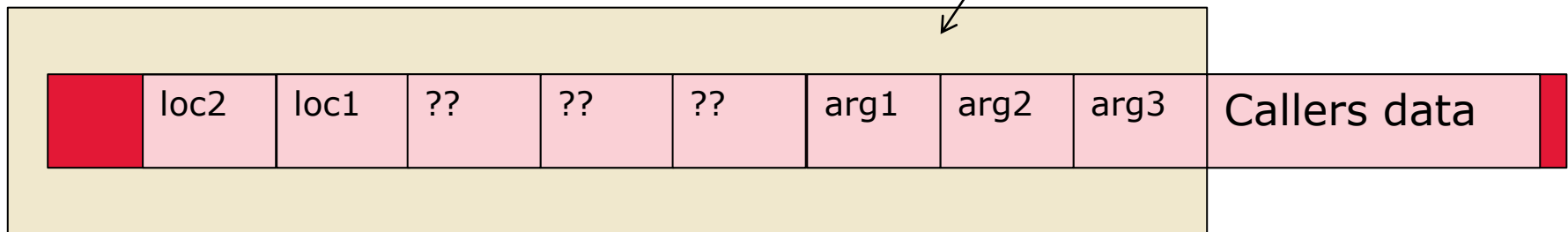


Locals pushed in same order as code (commonly) but compiler may choose different order based on optimisations

Accessing Variables

```
void func (char *arg1, int arg2, int arg3)
{
...
loc2++;
}
```

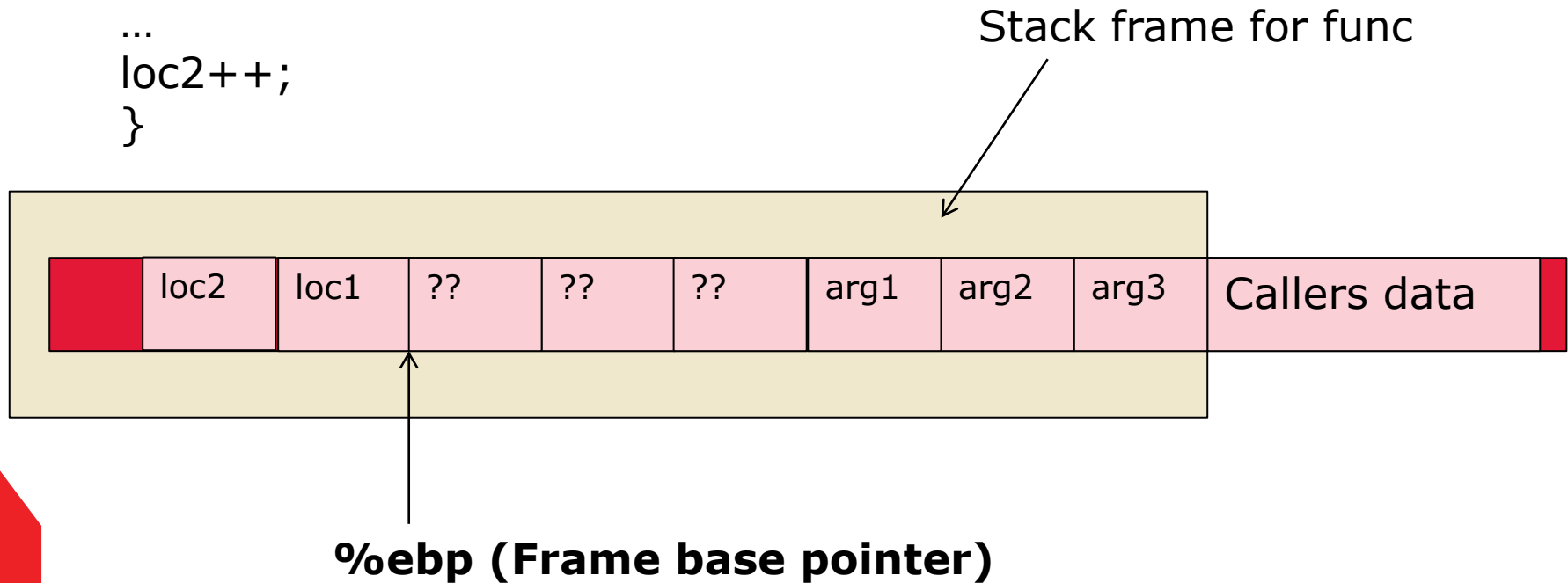
Stack frame for func



- How do we know where is loc2?
- We can't know at compile time
- But we know **relative address**: loc2 is 8 bytes before ??s or in other words 8 bytes before where %esp was at the start

Accessing Variables

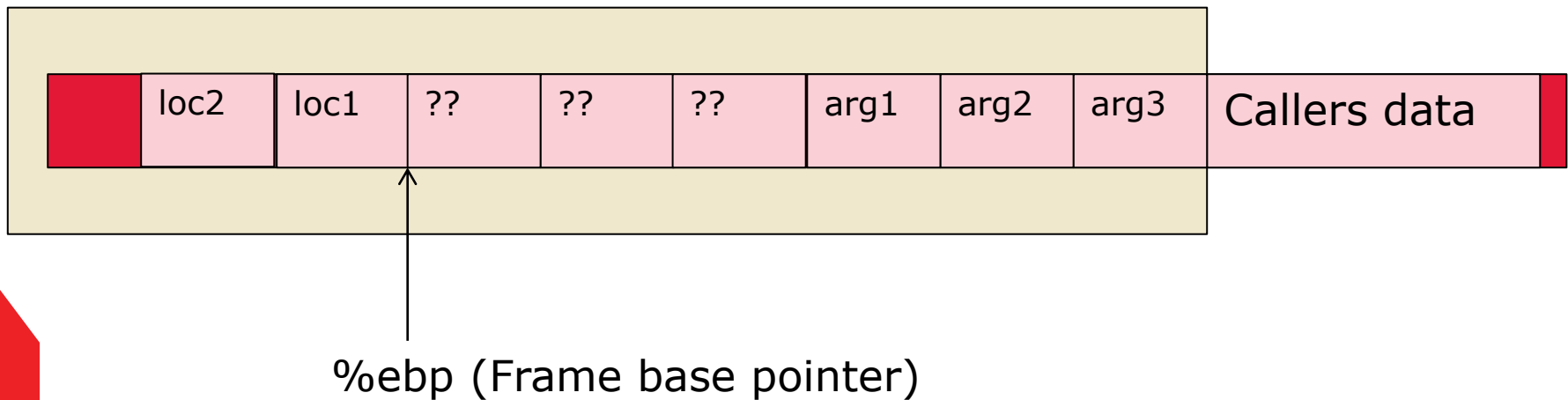
```
void func (char *arg1, int arg2, int arg3)
{
...
loc2++;
}
```



- %ebp is set to %esp at function entry
- So address of loc2 is 8 bytes before ??s or $-8(\%ebp)$

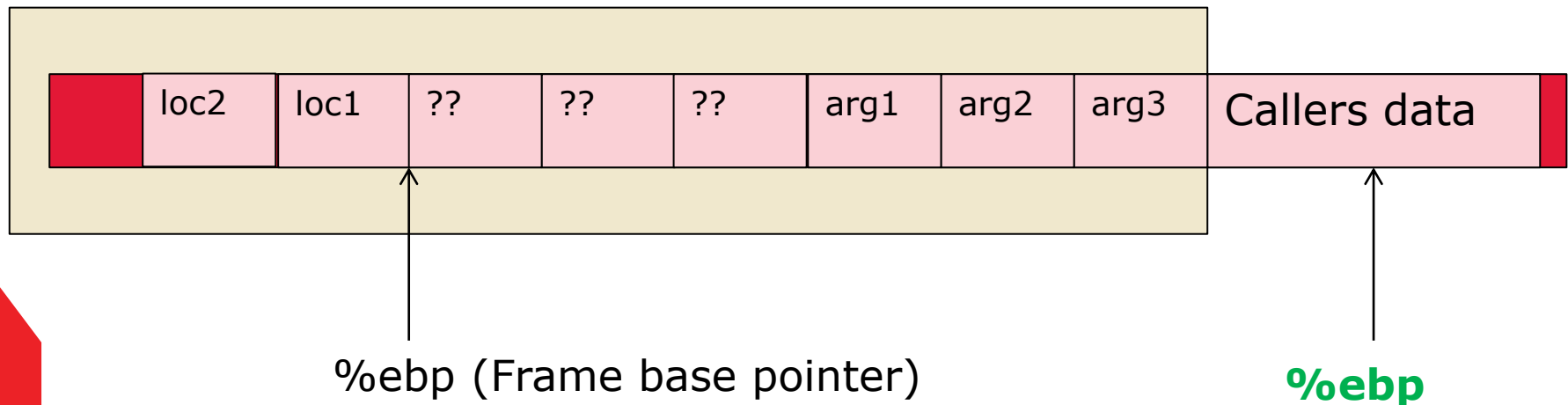
Returning from Functions

```
int main()  
{  
  func("blah",2,2);  
  ...  
}
```



Returning from Functions

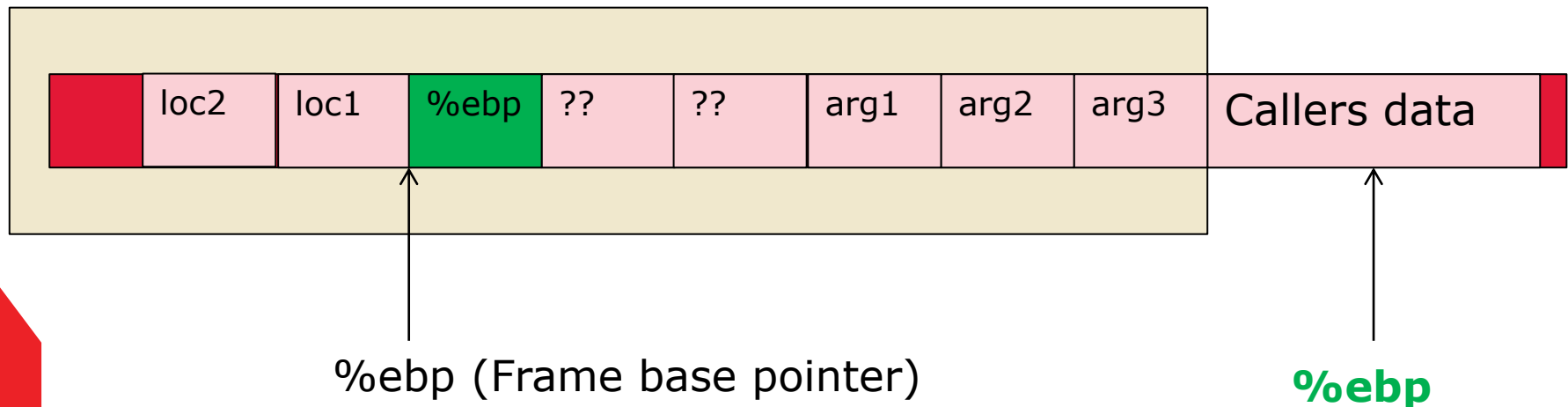
```
int main()  
{  
  func("blah",2,2);  
  ..  
}
```



Before we called "func", main was using frame base pointer for its own stack frame

Returning from Functions

```
int main()  
{  
  func("blah",2,2);  
  ...  
}
```



Frame base pointer of caller is pushed on stack before executing function so it can be restored on return

Returning from Functions

**How do we resume
where we left off when
we called func?**

Instruction pointer %eip
points to current instruction

%eip →

```
...  
0x4a7 mov $0x0,%eax  
0x4a2 call <func>  
0x49b movl $0x804.., (%esp)  
0x493 movl $0xa, 0x4(%esp)  
...
```

4G

0xffffffff

0

0x00000000

46

Text

Returning from Functions

How do we resume
where we left off when
we called func?

%eip →

```
...  
0x5bf mov %esp,%ebp  
0x5be push %ebp  
...
```

```
...  
0x4a7 mov $0x0,%eax  
0x4a2 call <func>  
0x49b movl $0x804.., (%esp)  
0x493 movl $0xa, 0x4(%esp)  
...
```

4G

0xffffffff

0

0x00000000

47

Text

Returning from Functions

**How do we resume
where we left off when
we called func?**

Instruction pointer %eip
points to current instruction

%eip →

```
...  
0x4a7 mov $0x0,%eax  
0x4a2 call <func>  
0x49b movl $0x804.., (%esp)  
0x493 movl $0xa, 0x4(%esp)  
...
```

4G

0xffffffff

0

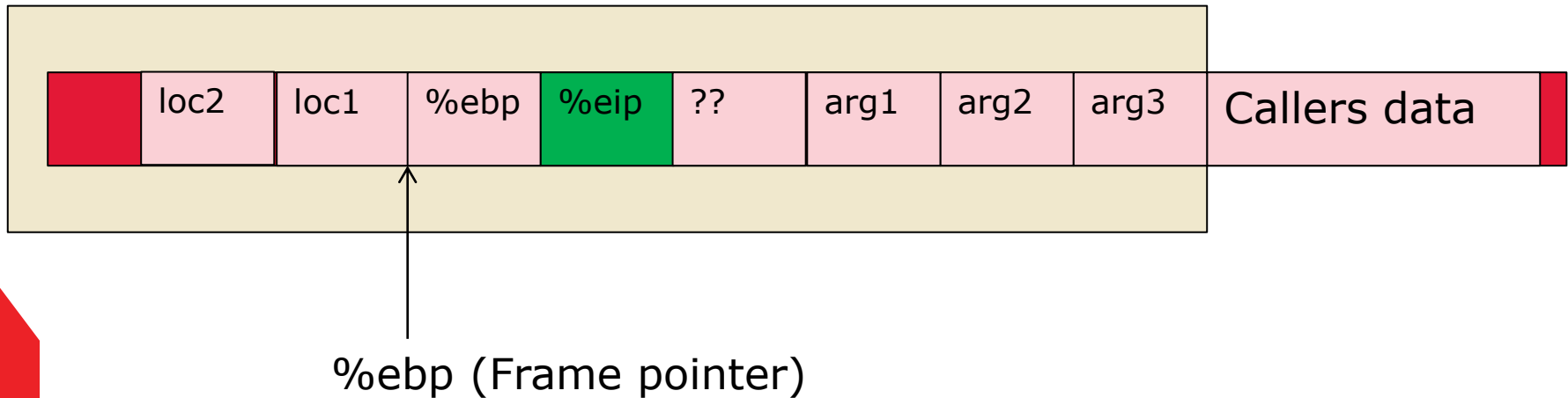
48

Text

0x00000000

Returning from functions

```
int main()  
{  
  func("blah",2,2);  
  ..  
}
```



We can do same trick and push instruction pointer %eip onto stack just before call

Stack and Function Summary

Calling function

Push args on stack

Push return address

Jump to func address

Called function

Push the old frame base ptr

Set frame base ptr to where end of stack is now

Push local vars on stack

Returning function

Reset previous stack frame $\%ebp = (\%ebp)$

Jump back to return address $\%eip = 4(\%ebp)$

64bit vs. 32bit Application

- Explanation on previous slides for 32bit apps
 - Simpler than 64bit
- Some things are different for 64bit
 - Larger address space
 - Different register names: RSP, RBP and RIP instead of ESP, EBP and EIP
 - Up to 6 args are passed via registers, only rest is pushed on stack

Buffer Overflows

Buffer: Contiguous memory associated with variable

Overflow: Put more data in it than we have room for :P

Where does Extra Data Go?

Now you are experts in memory layouts...

"Benign" Outcome

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

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

Upon return, %ebp is now
0x0021654d



"Benign" Outcome

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

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

<http://myitforum.com/myitforumwp/2012/08/24/sccm-2012-reporting-for-dummies-adding-the-reporting-point/>



00 00 00 00	%ebp	%eip	&arg1
m e ! \0			
A u t h	4d 65 21 00	%eip	&arg1

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);
    ...
}
```



buffer authenticated

00 00 00 00	00 00 00 00	%ebp	%eip	&arg1
m e ! \0				
A u t h	4d 65 21 00	%ebp 56	%eip	&arg1

Even worse?

00 00 00 00	00 00 00 00	%ebp	%eip	&arg1
-------------	-------------	------	------	-------

strcpy lets us write as much data as we want, so why stop here!

buffer

authenticated

00 00 00 00	00 00 00 00	%ebp	%eip	&arg1
-------------	-------------	------	------	-------

Even worse?

00 00 00 00	00 00 00 00	%ebp	%eip	&arg1
-------------	-------------	------	------	-------

strcpy lets us write as much data as we want, so why stop here!

buffer

authenticated



Code Injection



1. Load evil hacker code into memory
2. Somehow get it to run

Recap: What does %eip do?

Loading Code into Memory

- It must be machine code
- Can't contain any zero bytes – why?
 - Zero is terminator for C strings
- Depending on function exploited there may be other bad characters that can't be used
- It can't use stack (we are smashing that!)
- Much easier if user supplied strings are blindly accepted by program 😊

Getting Injected Code to Run

Calling function	Push args on stack
	Push return address
	Jump to func address
Called function	Push the old frame ptr
	Set frame ptr to where end of stack is now
	Push local vars on stack
Returning function	Reset previous stack frame <code>%ebp=(%ebp)</code>
	Jump back to return address <code>%eip=4(%ebp)</code>

Hijacking Saved %eip

00 00	00 00	%ebp	%eip	&arg1	Evil hacker code
-------	-------	------	------	-------	------------------------

If we overflow buffer far enough, we can overwrite %eip and make program jump to any address we like!

Buffer overflowed into %eip

00 00	00 00	00 00	F2 23	&arg1	Evil hacker code
			→		

0x F2 23

Hijacking Saved %eip

Buffer overflowed into %eip

00 00	00 00	00 00	F2 23	&arg1	Evil hacker code
			→		

0x F2 23

- Of course, it's harder in real life
 1. Without seeing code, we don't know exactly how far overflowed buffer is from saved %ebp or %eip
 2. We don't know absolute addresses of our hacker code and hitting wrong address will just cause crash

Getting Evil Code to Run

Problem 1: Without seeing code, we don't know exactly how far overflowed buffer is from saved %ebp or %eip

- We can find out by trial and error with debugger
- There are tools that create random pattern
- Inject pattern into buffer and look up EIP value
- Put pattern and EIP value into tool and it will tell you exact offset

Getting Evil Code to Run

Problem 2: We don't know absolute address of our hacker code

- What we do know is that at end of function ESP will point behind EIP after EIP is popped
- If we can jump to ESP we are almost there
- Locate "jmp esp" instruction in executable or library used at fixed address
- Overwrite EIP with address of "jmp esp" command
- So we will jump to "jump esp" command and from there to ESP which is roughly where evil code is
- Use **NOP sled** to overwrite anything from there to the code and slide into code

NOP Sled

Buffer overflowed into %eip

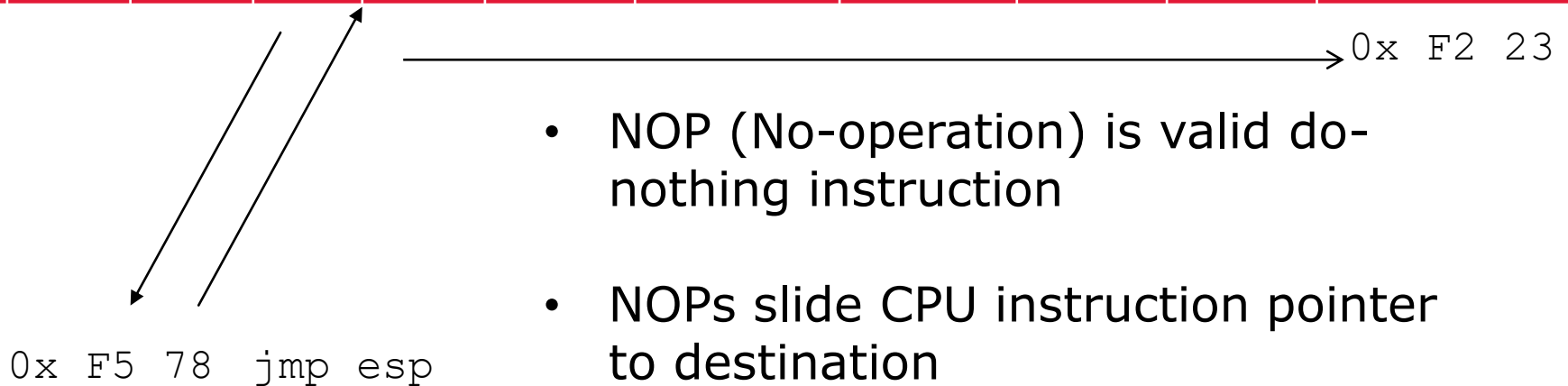


0x F2 23

0x F5 78 jmp esp

NOP Sled

Buffer overflowed into %eip



- NOP (No-operation) is valid doing nothing instruction
- NOPs slide CPU instruction pointer to destination
- Evil hacker code is executed

Mitigations

Memory and type safe languages do not suffer so much from these issues, but even with languages like C , there are some automatic defences

Stack Canaries

- These are small integer values, placed by compiler near %eip
- If these values are corrupted, then stack smashing is detected and caught
- But program will still crash...



<http://io9.gizmodo.com/why-did-they-put-canaries-in-coal-mines-1506887813?IR=T>

Non-executable Stack

- Enforce policy on stack memory region that disallows code execution from the stack
- Based on NX (No-eXecute) bit used in CPUs to segregate areas of memory for use by either storage of code or storage of data (Intel calls it XD bit)
- However, not difficult to store code in unprotected memory regions like heap, and so not very hard to circumvent this
- Often this is still not used for many applications

ASLR

- Another common technique is called Address Space Layout Randomization (ASLR)
- Ensures that layout of variables in memory is not consistent from run to run and therefore stack smashing attacks are much harder (but not impossible)



Breaking ASLR

- Researchers showed that utilising side channel in CPU's branch predictor allows to break ASLR
- In 2017 researchers found way to break ASLR via MMU cache interaction side channel that even works in JavaScript
 - Worse, they conclude that caching and strong address space randomization are mutually exclusive

Other Memory Attacks

- Stack smashing violates integrity and also availability, but may also violate confidentiality
- There are other attacks too...
- Heap overflow (buffers allocated by malloc() reside on 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 );      copy one into buff
    strcat( s->buff, two );      copy two into buff
    return s->cmp( s->buff, "file://foobar" );
}
```

Must have $\text{strlen}(\text{one}) + \text{strlen}(\text{two}) < \text{MAX_LEN}$
or we overwrite cmp function pointer

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);
    }
}
```

- Assume malloc() takes 32-bit unsigned int as argument so maximum size of allocation is 0xFFFFFFFF
- Assume nresp can also be up to 0xFFFFFFFF

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);
    }
}
```

- If nresp is 0x40000000 and assuming sizeof(char*)=4 we overflow 32bit and get malloc(0). Then the following loop will copy into unallocated memory

Integer Overflow Results

\$370 million integer overflow in Ariane 5 rocket
(<https://hownot2code.com/2016/09/02/a-space-error-370-million-for-an-integer-overflow/>)



Corrupting Data

- These attacks have so far corrupted code
- We could also corrupt data
 - Modify secret key
 - State variables
 - Change strings that are interpreted later (similar to SQL injection!)

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 (!isctrl(buf[i])) putchar(buf[i]);
            else putchar('.');
        printf("\n");
    }
}
```

Read integer
indicating
message length

Read message

Print message by
printing each
character up to
message length

Read Overflow

```
File Edit View Terminal Help
lulz $:./a.out
26
ICT287 Computer Security
ICT287 Computer Security..
█
```


Read Overflow

```
File Edit View Terminal Help
lulz $:./a.out
26
ICT287 Computer Security
ICT287 Computer Security..
38
Secret password is awesomepassword123
Secret password is awesomepassword123.
█
```

Read Overflow

```
File Edit View Terminal Help
lulz $:./a.out
26
ICT287 Computer Security
ICT287 Computer Security..
38
Secret password is awesomepassword123
Secret password is awesomepassword123.
█
```

So let's exploit this

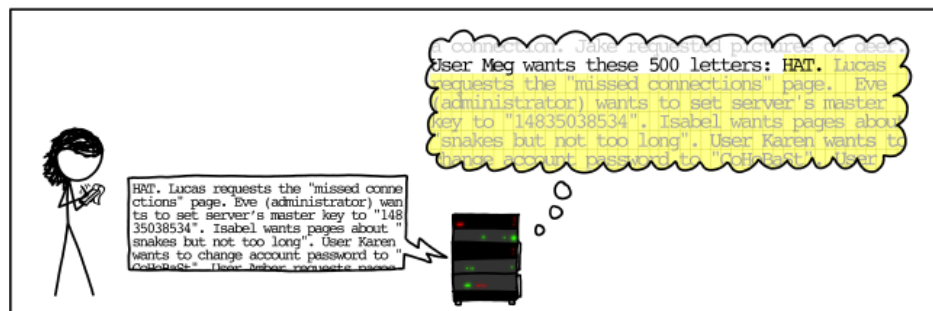
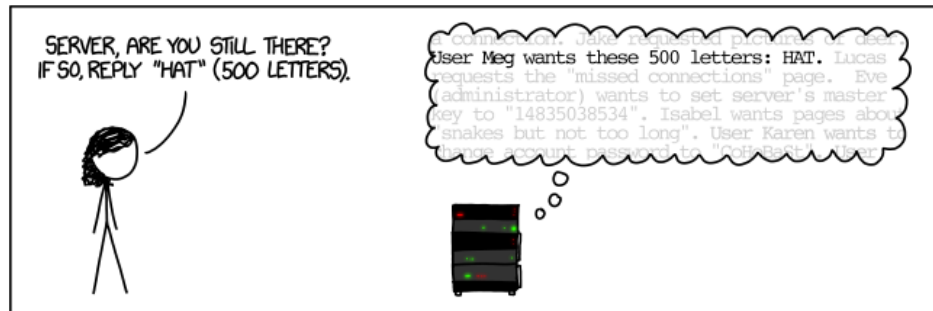
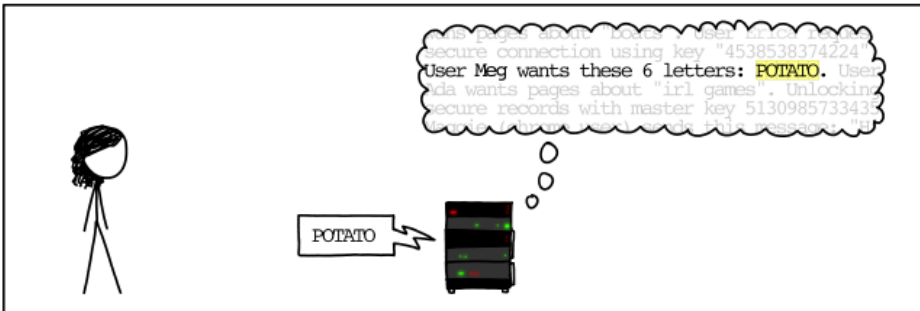
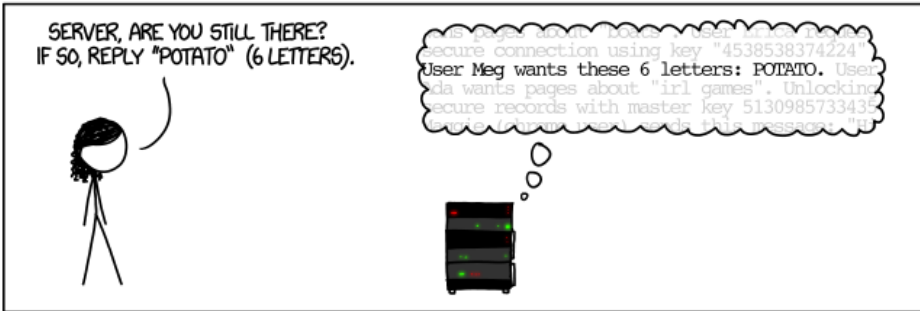
```
38
short string now
short string now.. awesomepassword123
█
```

Heartbleed

- With SSL/TLS computers send heartbeat requests, which is payload along with payload length
- Receiving computer then sends payload back to sender
- Affected versions of OpenSSL allocate memory buffer for return message based on length field in requesting message with no bounds checking
- Malicious user can craft heartbeat requests with incorrect length
- Causes potential buffer over-read



Heartbleed



<https://xkcd.com/1354/>

Null Pointer Dereference

- Accessing memory for pointer that does not point to anything (NULL)
- For example, if below `user_supplied_addr` does not resolve, `strcpy` will dereference NULL pointer `hp`

```
validate_addr_form(user_supplied_addr);  
addr = inet_addr(user_supplied_addr);  
hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);  
strcpy(hostname, hp->h_name);
```

- If attacker can control this, they can potentially bypass later logic, reveal debugging information or simply crash system/application (DoS)
- They might even be able to execute code

Null Pointer Dereference

- Some older OS (Windows 7) allow memory to be allocated at NULL page which in combination with dereference used to call function via function pointer enables code execution!

- Imagine following code

```
some_pointer = parse_user_input();  
some_pointer->callback();
```

- If attacker can make some_pointer=NULL, allocate NULL page and put instructions in NULL page, these instructions will be executed

Format String Attacks

- Format string is argument of format function that governs how some data is printed to terminal or in buffer, e.g. first argument of `printf()`
- Format string parameters like `%s` or `%d` define type conversion of data
- Example

```
printf("The magic number is: %d\n", var);
```

- Compiler will warn if number of format parameters is not same as number of variables
- But format string may not be known at compile time, so no way compiler can check properly!

Format String Attacks

- As usual arguments of printf() including format string are pushed on stack
- For each format string parameter printf() will fetch argument from stack
- Argument may be pointer to data (e.g. %s)
- Let's imagine we use printf() to print unsanitised user input
- Example

```
printf(user_input);
```


Format String Attacks

- `user_input = "%s%s%s%s%s%s%s%s%s"`
 - Will likely crash program as `printf()` will access various memory addresses, some of which are likely not allocated...
- `user_input = "%08x %08x %08x %08x %08x";`
 - Will print five values from stack as hexadecimal with 8 digit padding
 - OK, what's harm of printing some random stuff from stack?

Format String Attacks

- Can print memory contents from specified location
- Say we want to print memory at location 0x10014808
- We just need to include this address in format string which will be stored on stack
- Note that \xVV will store byte with value VV
- Don't know exactly where it will be stored in relation to printf() stack pointer
- Use %x to slide pointer to address stored on stack (bit like NOP sled), then print address
- `user_input = "\x10\x01\x48\x08 %x %x %x %s";`

Lecture Summary and Week Ahead

- Common Vulnerabilities and Exposures (CVE)
- Common Weakness Enumeration (CWE)
- Layout of program memory
- Buffer overflow attacks
- Variants: heap overflow, read overflow, integer overflow
- Format string attacks

- Labs: Buffer overflows
- Next week: Malware