case study

E-VOTING ANALYSIS

Kohno et al., IEEE S&P 2004

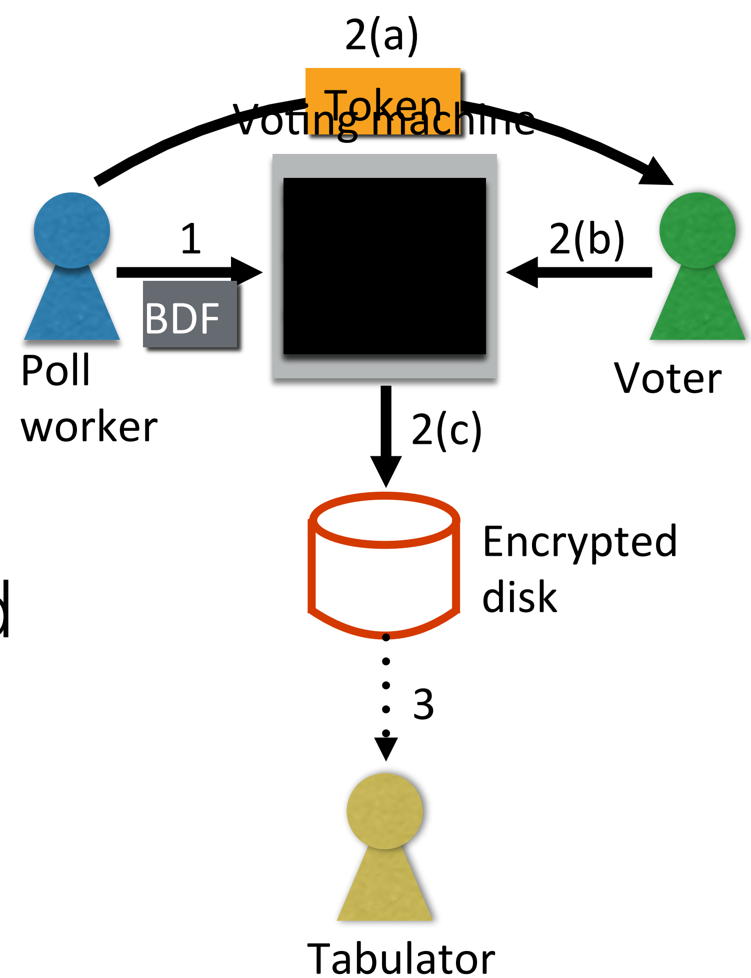
Halderman, 2016

“Security mindset”

- Consider a complex system:
  - Potential security threats?
  - Hidden and explicit assumptions
  - How to mitigate the risks?
  - What are different players’ incentives?

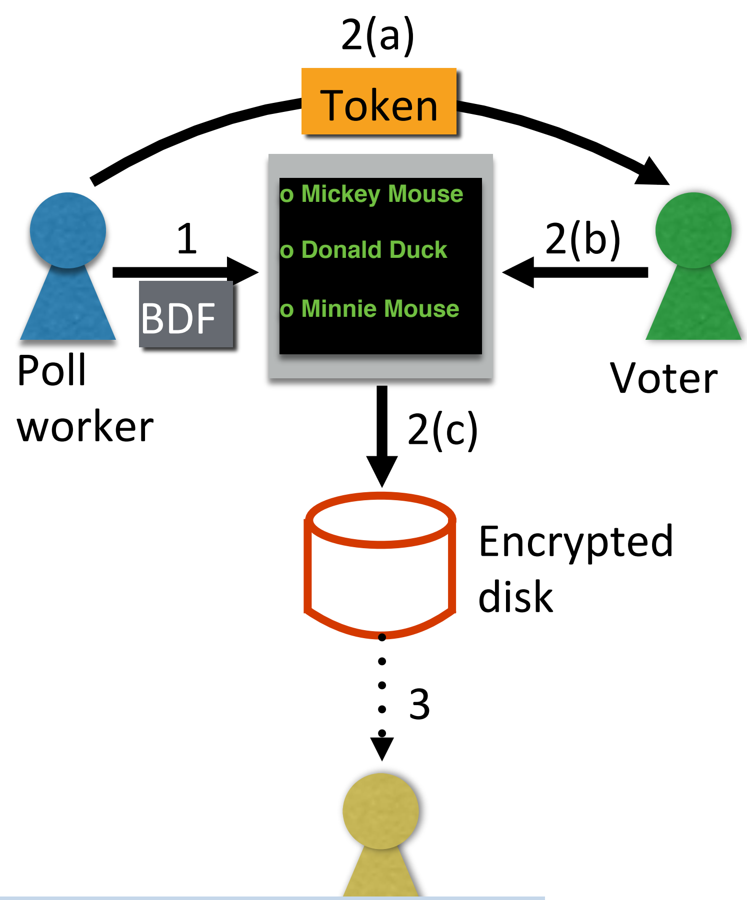
1. Summarize the system

1. Pre-election: Poll worker loads "ballot definition" via e.g. USB
2. Voting: Voter obtains single-use smartcard, votes, vote stored encrypted, card canceled
3. Post-election: Votes decrypted and sent to tabulator, who counts



2. Identify goals/requirements

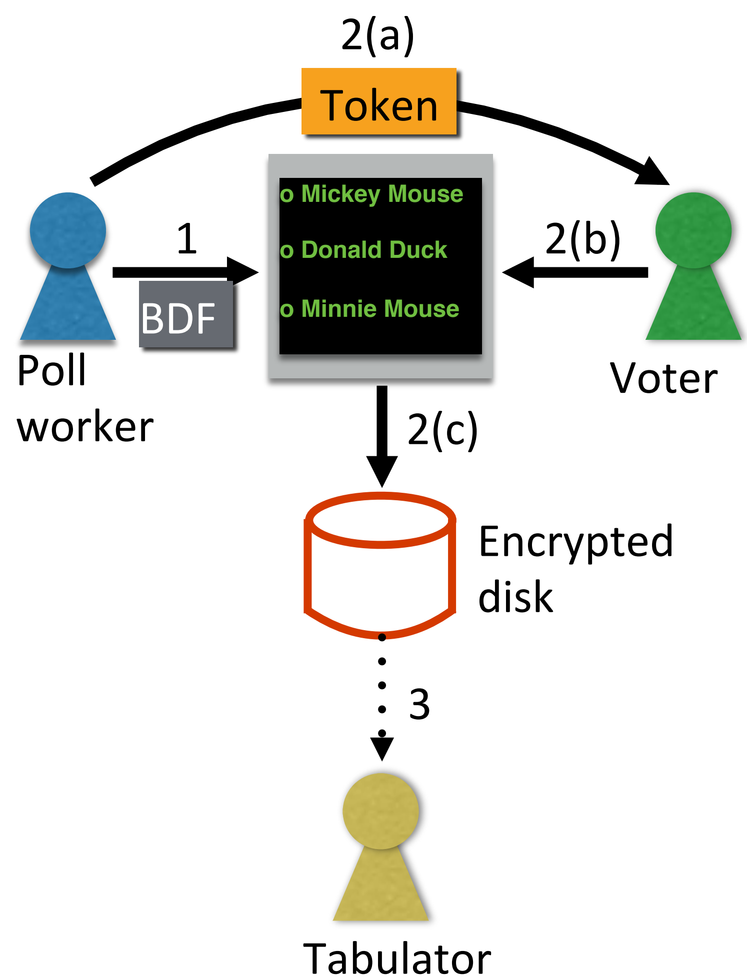
- **Confidentiality**: Can't find out who I voted for
- **Integrity**: Can't alter votes
- **Availability**: Can't deny opportunity to vote
- **Usability**: General public can vote correctly without undue b[urden]



What if the attacker can violate these, but you catch him/her?

3. Identify adversaries/threats

- Poll worker, voter, outsider
- Display one vote / count a different vote
- Vote multiple times
- End election early (DOS)
- Tamper with stored data
- Reveal who voted for whom



Diebold Accuvote TS

- Used in 37 states! (in 2004)
- No cryptography protects smartcards, ballot definition file
- “Protected counter” in single, mutable file
- Pose as voting machine, send to tabulator
- Homebrew crypto protects vote logs
  - Hardcoded key since at least 1998
- Read the paper for more

Follow-up

- More researchers confirmed these bugs and found others (got real hardware)
- State investigations: MD, CA, OH
  - Similar problems from other manufacturers
  - Sequoia AVC: designed 1980, used in NJ 2009
- "By the 2014 general election, 70% of American voters were casting ballots on paper"

Takeaways

- Adversarial thinking
- Whole-systems view
  - Hardware, software, network, users, economics
- Only as strong as weakest link
  - Break into building vs. sniff unencrypted traffic
  - You have to be right always, adversary once
- Never homebrew crypto!
- Security through obscurity DOESN'T WORK!

This time

We will begin our 1st section:
Software Security

By investigating
Buffer overflows
and other memory safety vulnerabilities

- History
- Memory layouts
- Buffer overflow fundamentals

screensaver --prompt=“Don’t unlock plz”

Don't unlock plz

Locked by dml

press ctrl-c to logout

screensaver --prompt=“Don’t unlock pretty plz”

Don't unlock pretty
plz

Locked by dml
press ctrl-c to logout

screensaver --prompt=“Don’t unlock plz ${}^{S}{}_{P}$ ${}^{S}{}_{P}$ ${}^{S}{}_{P}$ \
${}^{S}{}_{P}$ ${}^{S}{}_{P}$”

Don't unlock plz

Locked by dml

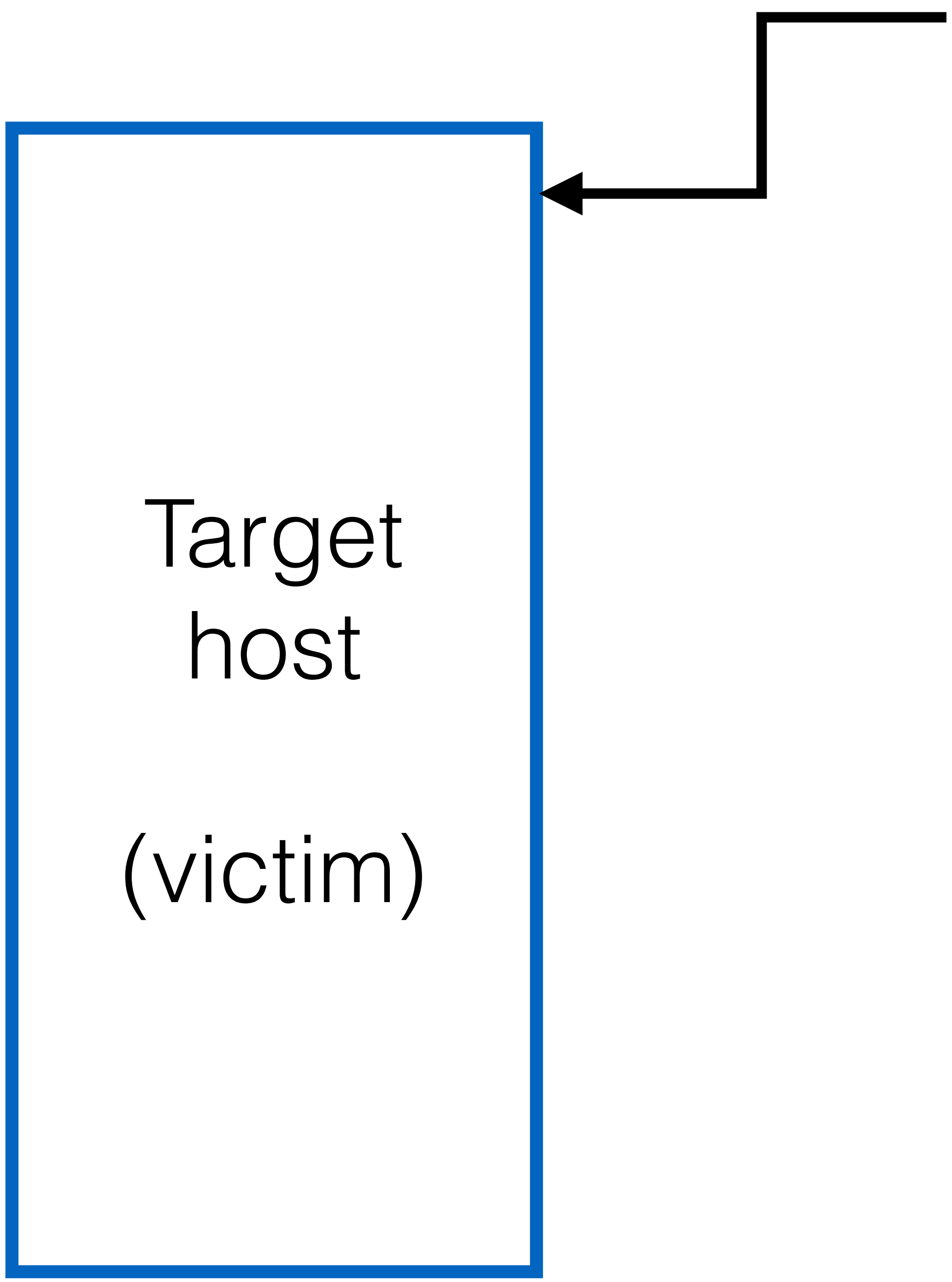
press ctrl-c to logout

```
screensaver –prompt=“Under maintenance;\
Do not interrupt␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠\
␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠␠”
```

```
Under maintenance;
Do not interrupt
```

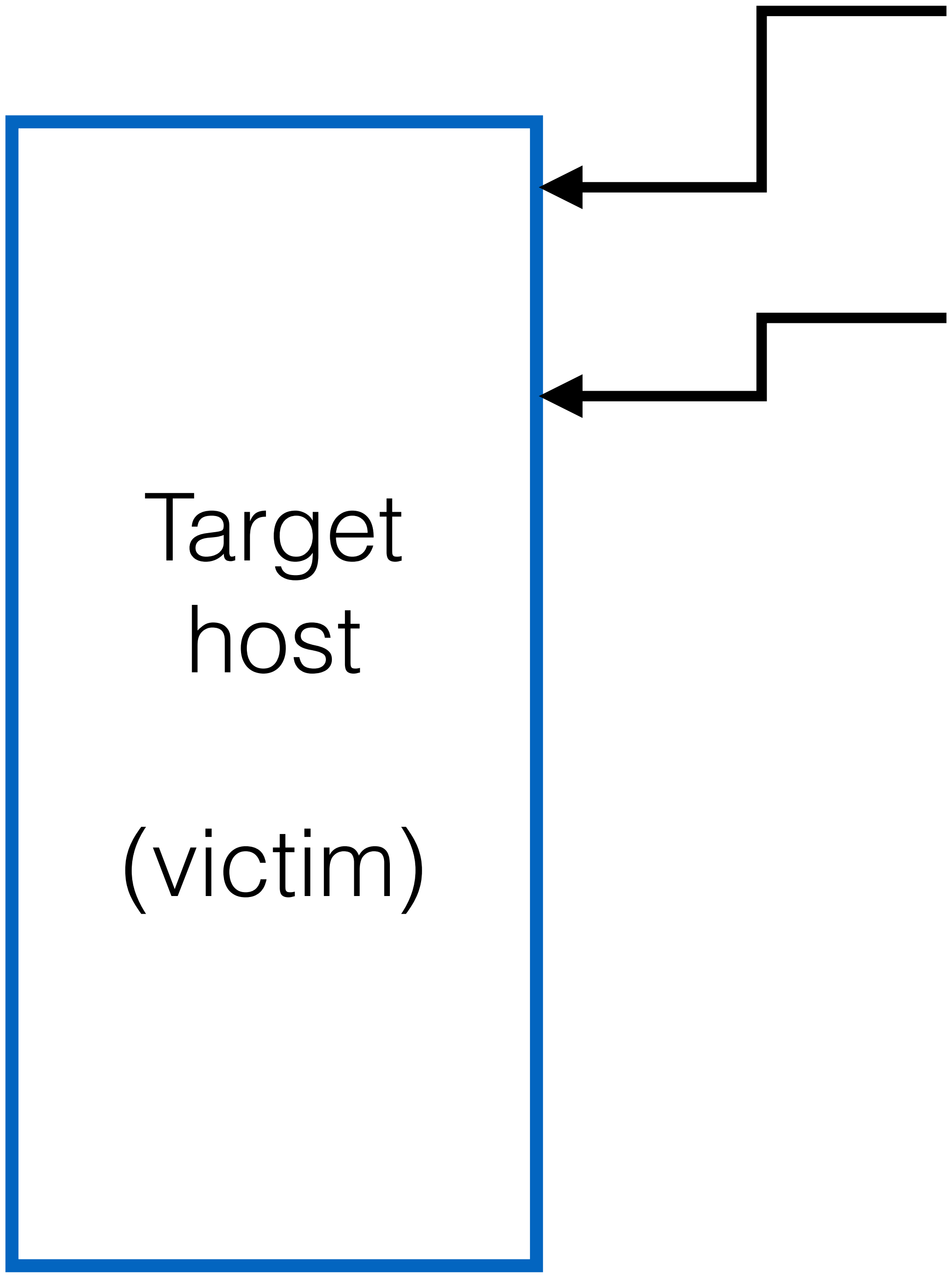
```
Locked by dml
press ctrl-c to logout
```

Most (interesting) software takes input



Direct user interaction
- command line interface (stdin)
- user opens a document

Most (interesting) software takes input



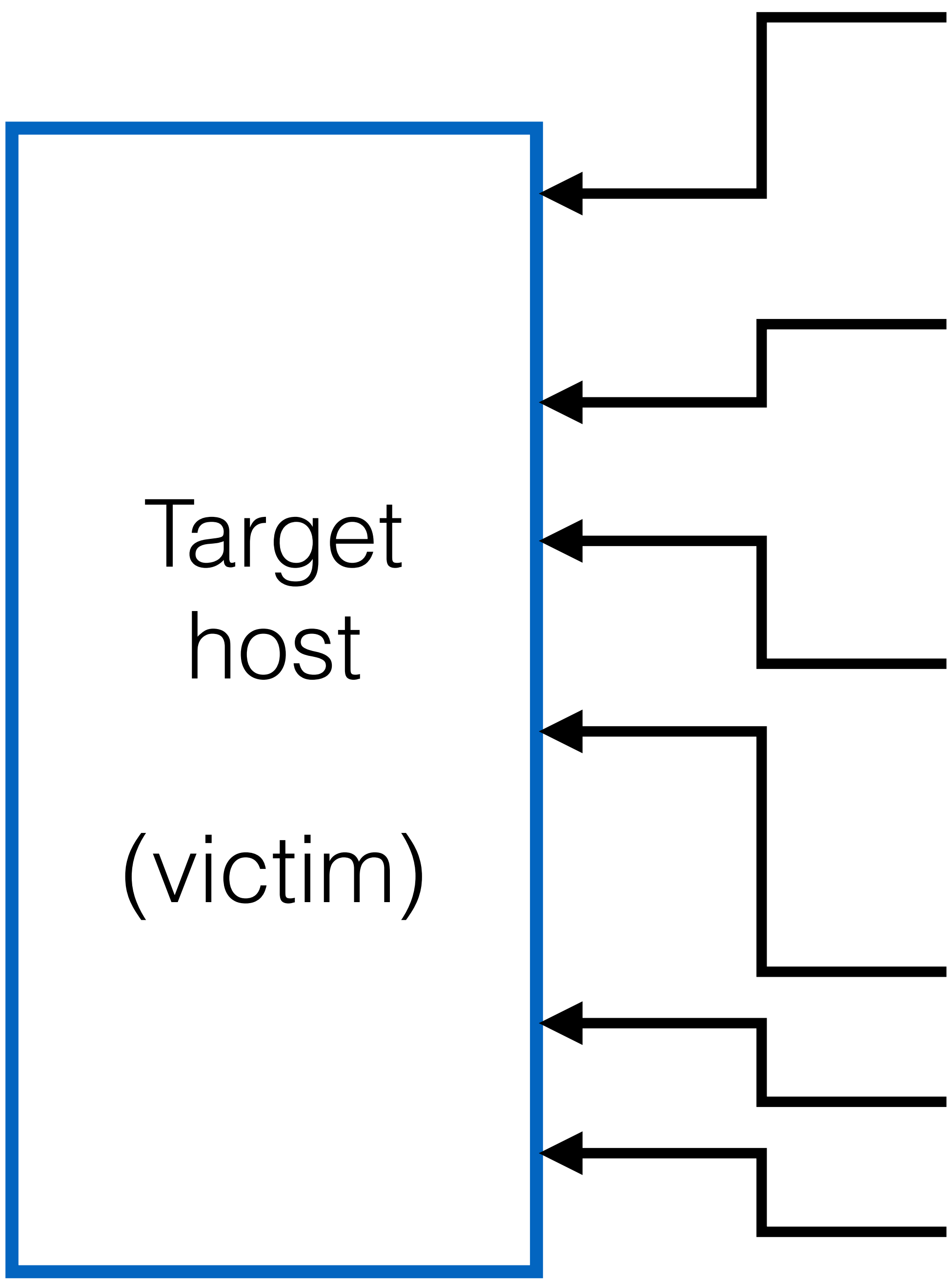
Direct user interaction

- command line interface (stdin)
- user opens a document

Network communication

- emails
- various protocols

Most (interesting) software takes input



Direct user interaction
- command line interface (stdin)
- user opens a document

Network communication
- emails
- various protocols

Sensing the outside world
- QR codes (to link w/ malware)
- sound recordings

Third-party libraries

Future code updates

Others…

Goal: Correct operation despite malicious inputs

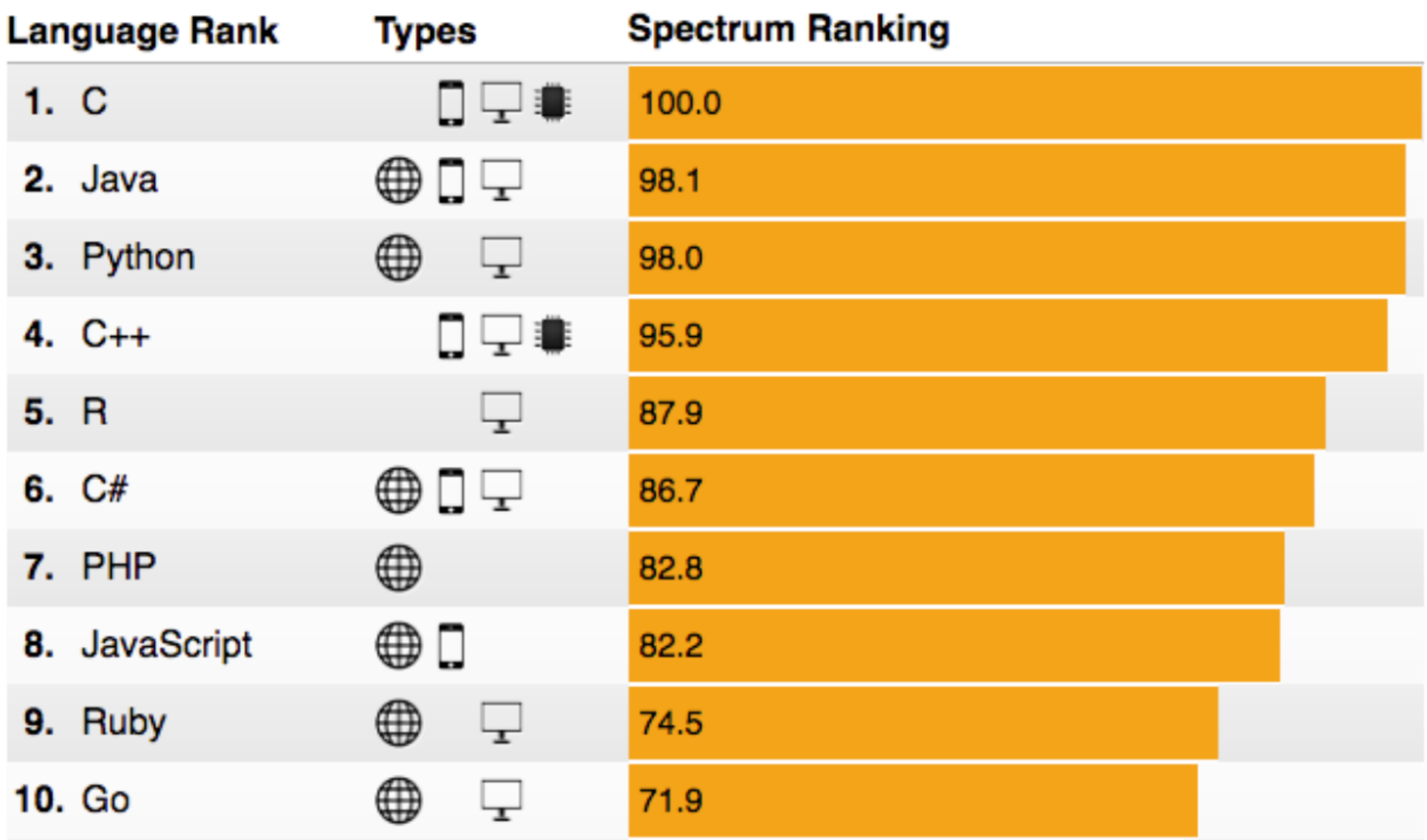
What is a buffer overflow?

- A **low-level** bug, typically in **C/C++**
  - Significant security implications!
- If accidentally triggered, causes a crash
- If maliciously triggered, can be **much worse**
  - **Steal** private info
  - **Corrupt** important info
  - **Run** arbitrary code

Why study them?

- Buffer overflows are still **relevant** today
  - C and C++ are still popular
  - Buffer overflows still occur with regularity
- They have a **long history**
  - Many different approaches developed to defend against them, and bugs like them
- They share **common features** with other bugs we will study
  - In **how the attack works**
  - In **how to defend against it**

C and C++ still very popular



http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

Critical systems in C/C++

- Most **OS kernels** and utilities
  - fingerd, X windows server, shell
- Many **high-performance servers**
  - Microsoft IIS, Apache httpd, nginx
  - Microsoft SQL server, MySQL, redis, memcached
- Ma
  - Mars rover, industrial control systems, automobiles, healthcare devices

A successful attack on these systems is particularly dangerous!

We're going to focus on C

A breeding ground for *buffer overflow attacks*

1988 1999 2000 2001 2002 2003

- Morris worm
  - Propagated across machines (too aggressively, thanks to a bug)
  - One way it propagated was a **buffer overflow attack** against a vulnerable version of `fingerd` on VAXes
    - Sent a special string to the finger daemon, which caused it to execute code that created a new worm copy
    - Didn't check OS: caused Suns running BSD to crash
  - End result: $10-100M in damages, probation, community service

(Robert Morris is now a professor at MIT)

We’re going to focus on C

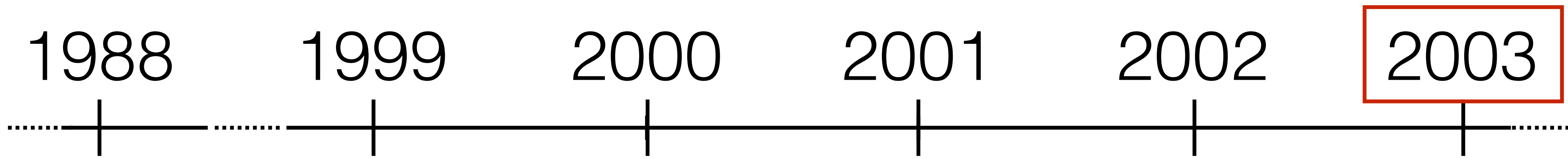
A breeding ground for *buffer overflow attacks*

1988 1999 2000 2001 2002 2003

- CodeRed
  - Exploited an **overflow** in the MS-IIS server
  - 300,000 machines infected in 14 hours

We’re going to focus on C

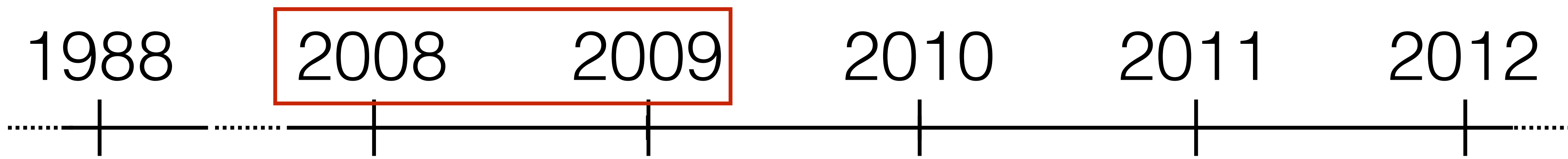
A breeding ground for *buffer overflow attacks*



- SQL Slammer
  - Exploited an **overflow** in the MS-SQL server
  - 75,000 machines infected in 10 *minutes*

We're going to focus on C

A breeding ground for *buffer overflow attacks*



- Conficker worm
  - Exploited an **overflow** in Windows RPC
  - ~10 million machines infected

We're going to focus on C

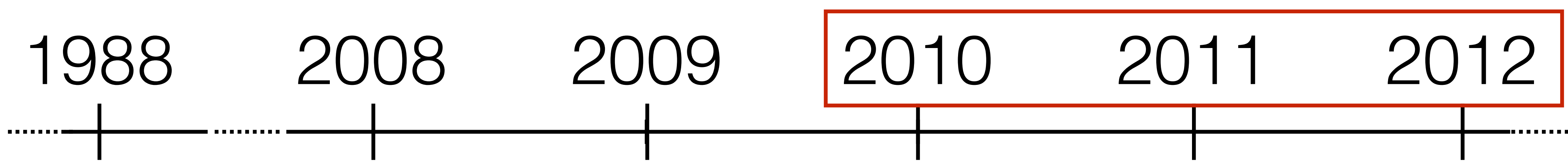
A breeding ground for *buffer overflow attacks*

1988 2008 2009 2010 2011 2012

- Stuxnet
  - Exploited several **overflows** nobody had at the time known about ("zero-day")
    - Windows print spooler service
    - Windows LNK shortcut display
    - Windows task scheduler
  - Also exploited the same Windows RPC overflow as Conficker
  - Impact: legitimized cyber warfare (*more on this later*)

We’re going to focus on C

A breeding ground for *buffer overflow attacks*



- Flame
  - Same print spooler and LNK **overflows** as Stuxnet
  - Cyber-espionage virus

Slashdot

Chann

stories

submissions

popular

blog

ask slashdot

book reviews

games

idle

yro

technology

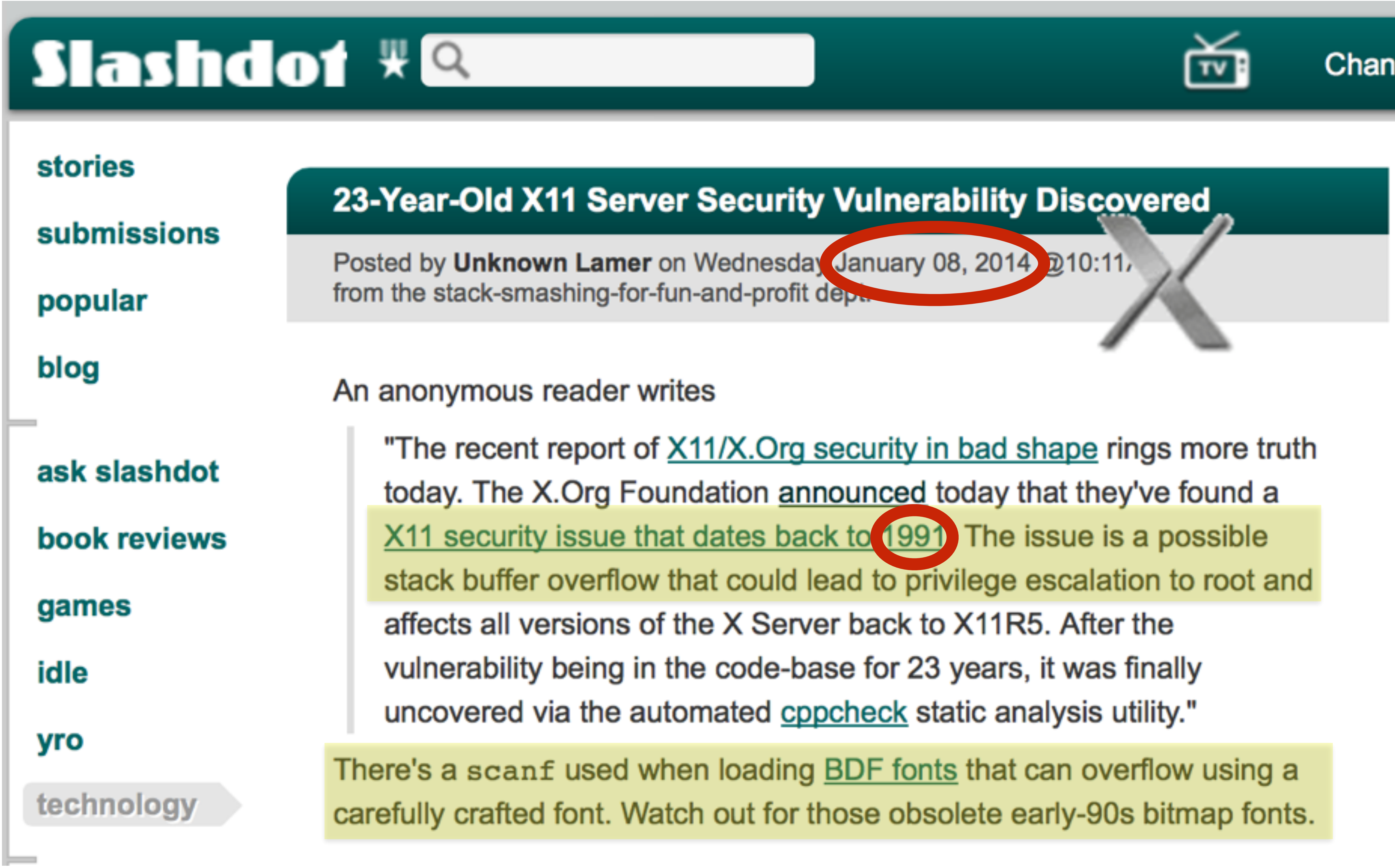
23-Year-Old X11 Server Security Vulnerability Discovered

Posted by **Unknown Lamer** on Wednesday January 08, 2014 @10:11
from the stack-smashing-for-fun-and-profit dept.

An anonymous reader writes

> "The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a `scanf` used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts.


Slashdot

Chann

stories

submissions

popular

blog

ask slashdot

book reviews

games

idle

yro

technology

23-Year-Old X11 Server Security Vulnerability Discovered

Posted by **Unknown Lamer** on Wednesday January 08, 2014 @10:11
from the stack-smashing-for-fun-and-profit dept.

An anonymous reader writes

> "The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a `scanf` used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts.

GHOST: glibc vulnerability introduced in 2000, only just announced last year

syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
  - Useful: one process issues the log messages, syslogd handles storing/disseminating them

```
void
add_lockdown_session(int fd)
{
        dispatch_once(&watch_init_once, ^{
                watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
        });

        dispatch_async(watch_queue, ^{
                if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;

                global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                                       global.lockdown_session_count + 1 * sizeof(int));

                if (global.lockdown_session_fds == NULL)
                {
                        asldebug("add_lockdown_session: realloc failed\n");
                        global.lockdown_session_count = 0;
                }
                else
                {
                        global.lockdown_session_fds[global.lockdown_session_count++] = fd;
                }

                global.watchers_active = direct_watch_count + global.lockdown_session_count;
        });
}
```

syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
  - Useful: one process issues the log messages, syslogd handles storing/disseminating them

```
void
add_lockdown_session(int fd)
{
        dispatch_once(&watch_init_once, ^{
                watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
        });

        dispatch_async(watch_queue, ^{
                if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;

                global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                                       global.lockdown_session_count + 1 * sizeof(int));

                if (global.lockdown_session_fds == NULL)
                {
                        asldebug("add_lockdown_session: realloc failed\n");
                        global.lockdown_session_count = 0;
                }
                else
                {
                        global.lockdown_session_fds[global.lockdown_session_count++] = fd;
                }

                global.watchers_active = direct_watch_count + global.lockdown_session_count;
        });
}
```

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                        global.lockdown_session_count + 1 * sizeof(int));
```

```c
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                global.lockdown_session_count + 1 * sizeof(int));
```

Array of `int`'s

```c
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                       global.lockdown_session_count + 1 * sizeof(int));
```

Array of `int`'s

Had this many `int`'s

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                        global.lockdown_session_count + 1 * sizeof(int));
```

Array of **int**’s

Want this many **int**’s

Takes *bytes* as 2nd arg

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                       global.lockdown_session_count + 1 * sizeof(int));
```

Array of **`int`**'s

Want this many **`int`**'s

Takes *bytes* as 2nd arg

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                       global.lockdown_session_count + 1 * sizeof(int));
```

Array of `int`'s

Want this many `int`'s

How many *bytes* should `global.lockdown_session_fds` be?

Takes *bytes* as 2nd arg

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                              global.lockdown_session_count + 1 * sizeof(int));
```

Array of **`int`**'s

Want this many **`int`**'s

How many *bytes* should `global.lockdown_session_fds` be?

```
global.lockdown_session_count + 1 * sizeof(int)
```

Takes *bytes* as 2nd arg

```
global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                       global.lockdown_session_count + 1 * sizeof(int));
```

Array of `int`'s

Want this many `int`'s

How many *bytes* should `global.lockdown_session_fds` be?

```
global.lockdown_session_count + 1 * sizeof(int)
```

```
(global.lockdown_session_count + 1) * sizeof(int)
```

syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
  - Useful: one process issues the log messages, syslogd handles storing/disseminating them

Buffer too small

```
void
add_lockdown_session(int fd)
{
        dispatch_once(&watch_init_once, ^{
                watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
        });

        dispatch_async(watch_queue, ^{
                if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;

                global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                                       global.lockdown_session_count + 1 * sizeof(int));

                if (global.lockdown_session_fds == NULL)
                {
                        asldebug("add_lockdown_session: realloc failed\n");
                        global.lockdown_session_count = 0;
                }
                else
                {
                        global.lockdown_session_fds[global.lockdown_session_count++] = fd;
                }

                global.watchers_active = direct_watch_count + global.lockdown_session_count;
        });
}
```

syslogd bug in Mac OS X & iOS

- syslog: message logging infrastructure
  - Useful: one process issues the log messages, syslogd handles storing/disseminating them

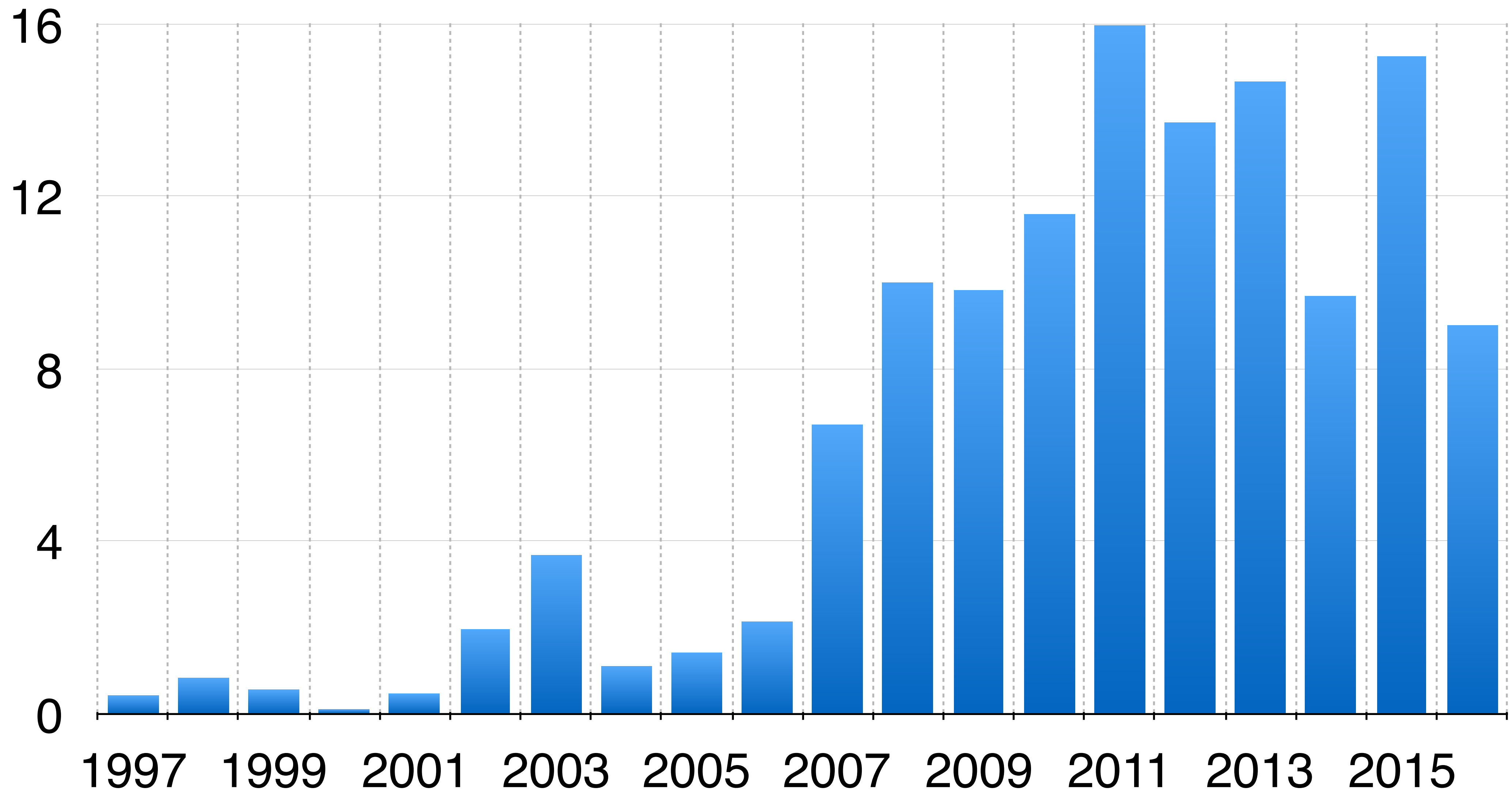
```
void
add_lockdown_session(int fd)
{
        dispatch_once(&watch_init_once, ^{
                watch_queue = dispatch_queue_create("Direct Watch Queue", NULL);
        });

        dispatch_async(watch_queue, ^{
                if (global.lockdown_session_count == 0) global.lockdown_session_fds = NULL;

                global.lockdown_session_fds = reallocf(global.lockdown_session_fds,
                                                       global.lockdown_session_count + 1 * sizeof(int));

                if (global.lockdown_session_fds == NULL)
                {
                        asldebug("add_lockdown_session: realloc failed\n");
                        global.lockdown_session_count = 0;
                }
                else
                {
                        global.lockdown_session_fds[global.lockdown_session_count++] = fd;
                }

                global.watchers_active = direct_watch_count + global.lockdown_session_count;
        });
}
```

Buffer too small

Writes beyond the buffer

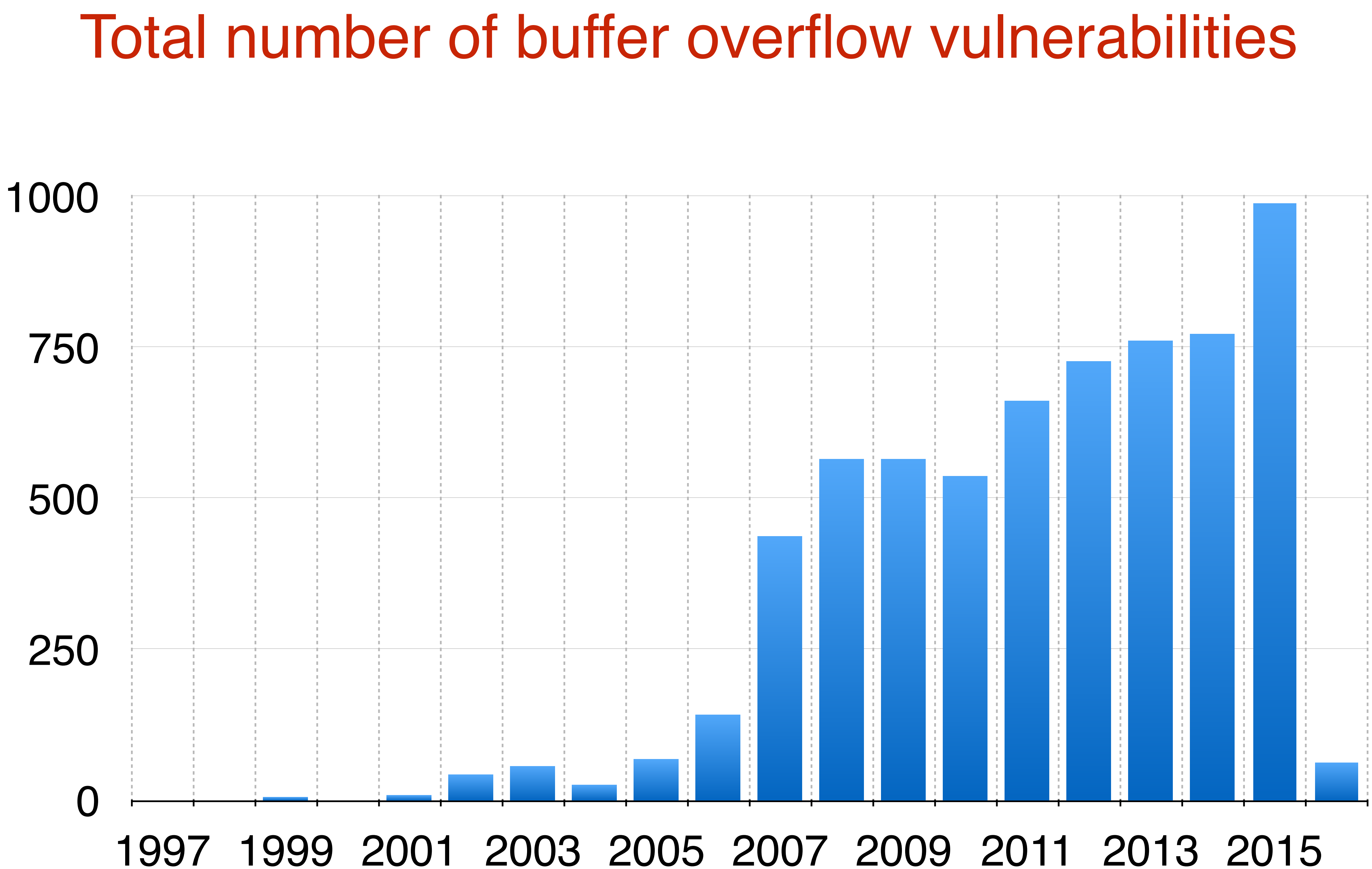
Buffer overflows are prevalent

Significant percent of *all* vulnerabilities



[Data from the National Vulnerability Database]

Buffer overflows are prevalent



Data from the National Vulnerability Database

Buffer overflows are impactful

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)

This class

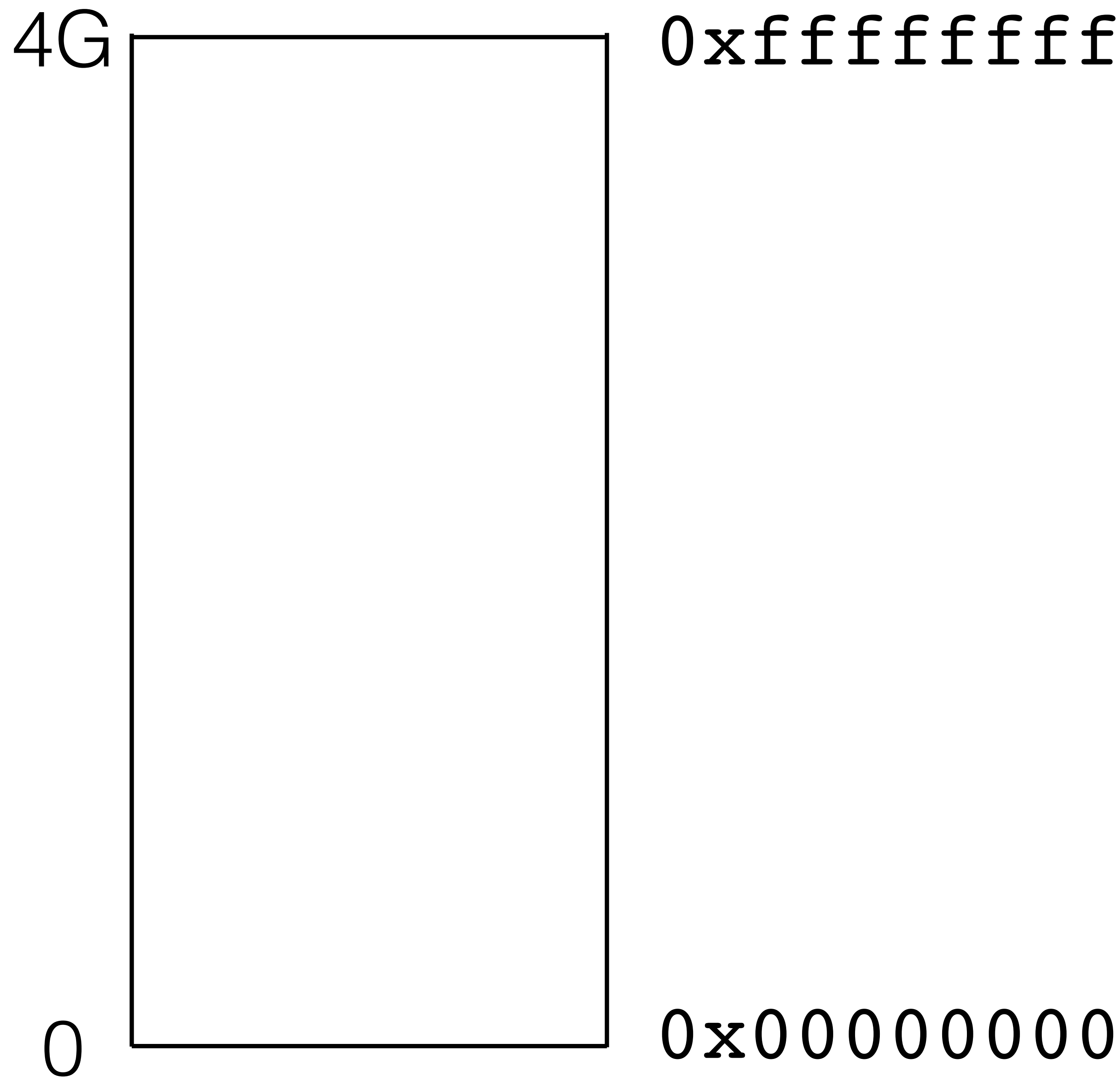
[MITRE's top-25 most dangerous software errors (from 2011)]

Note about terminology

- We will use **buffer overflow** to mean ***any access of a buffer outside of its allotted bounds***
  - An over-*read*, or an over-*write*
  - During *iteration* (“running off the end”) or by *direct access*
  - Could be to addresses that *precede* or *follow* the buffer
- Other terms you may hear (more specific)
  - *Underflow*, *over-read*, *out-of-bounds access*, etc.
  - Some use *buffer overflow* only for writing off the end

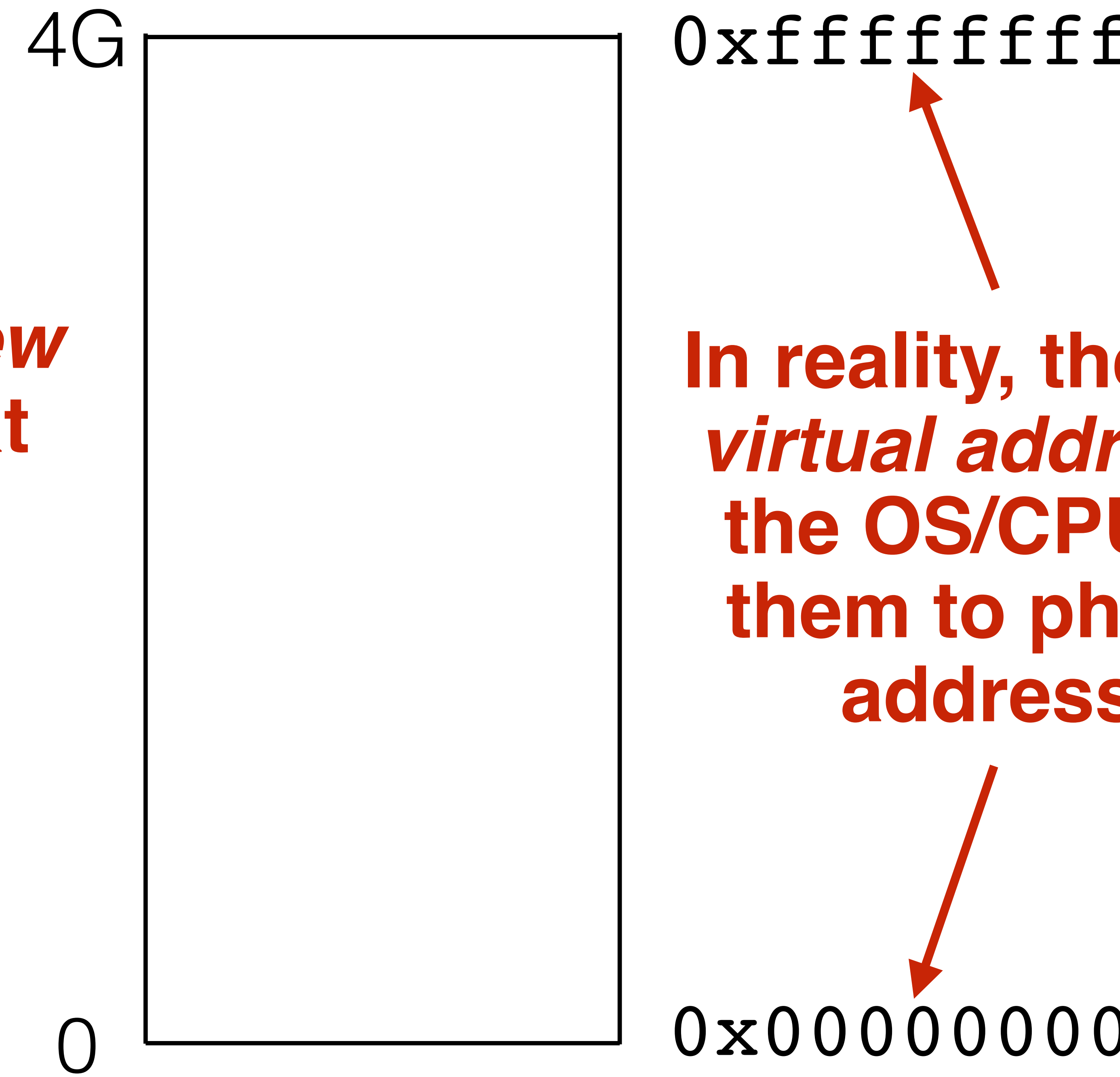
Memory layout

All programs are stored in memory



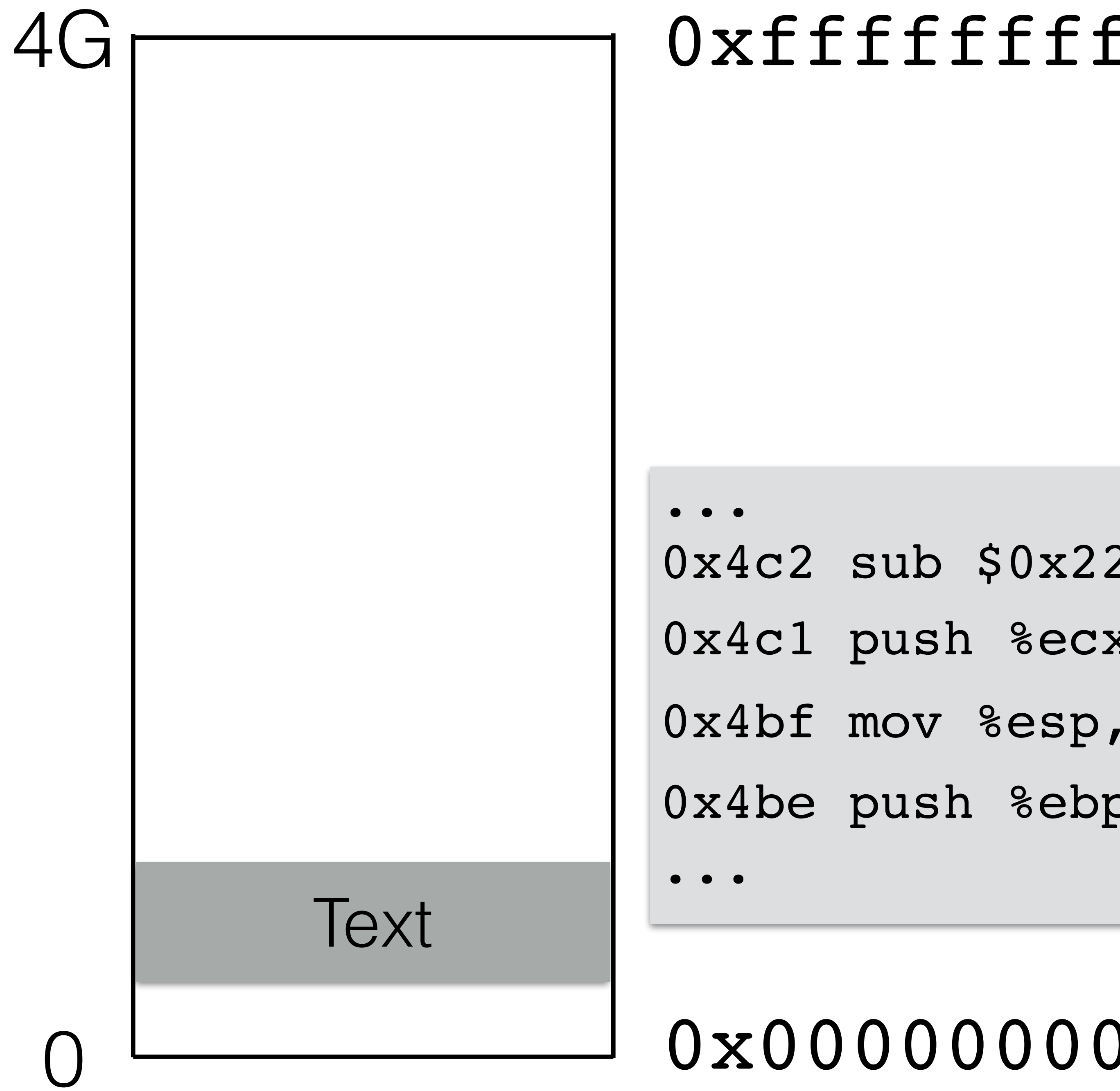
All programs are stored in memory

The *process's view* of memory is that it owns all of it



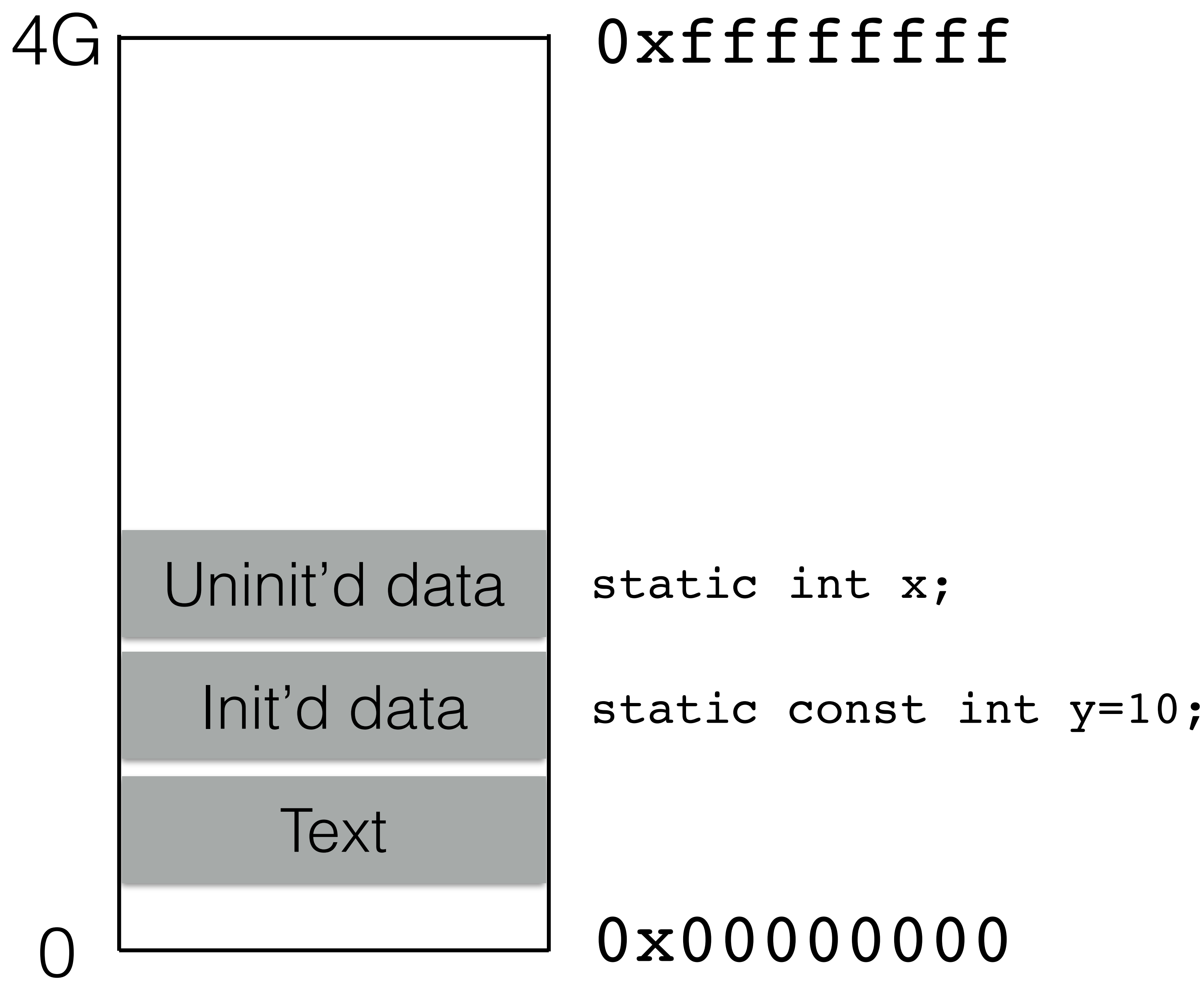
In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

The instructions themselves are in memory

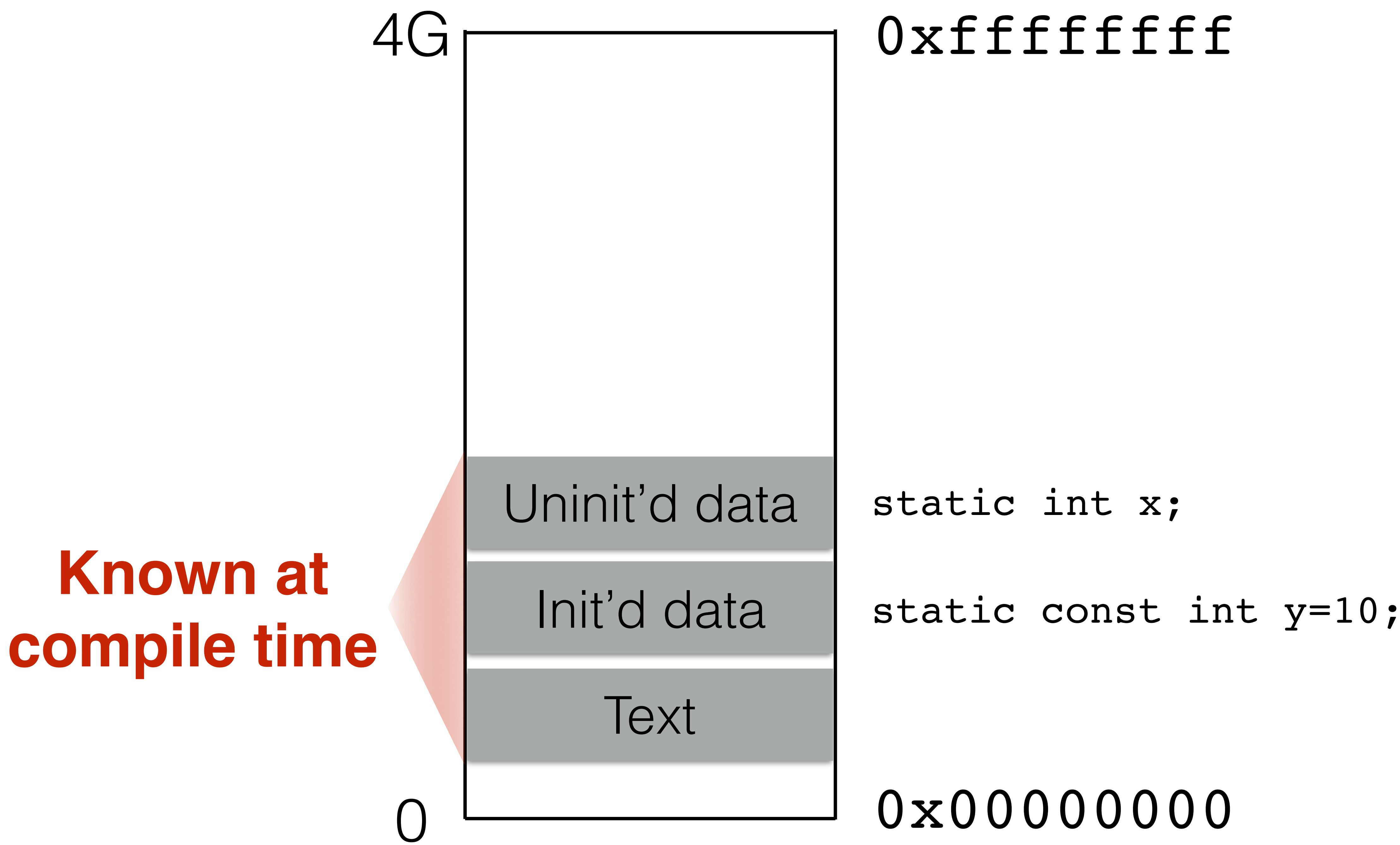



```
...
0x4c2 sub $0x224,%esp
0x4c1 push %ecx
0x4bf mov %esp,%ebp
0x4be push %ebp
...
```

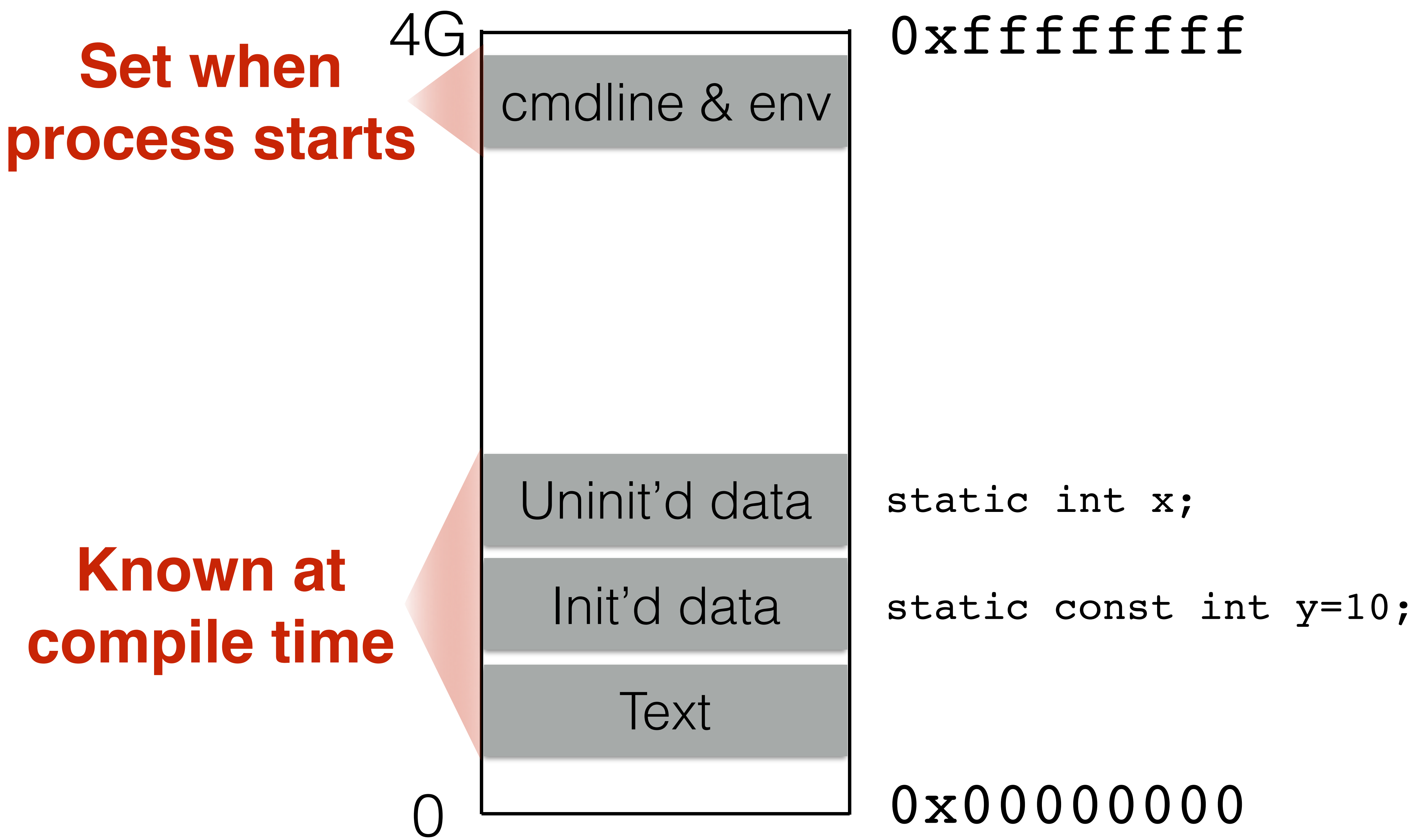
Data's location depends on how it's created



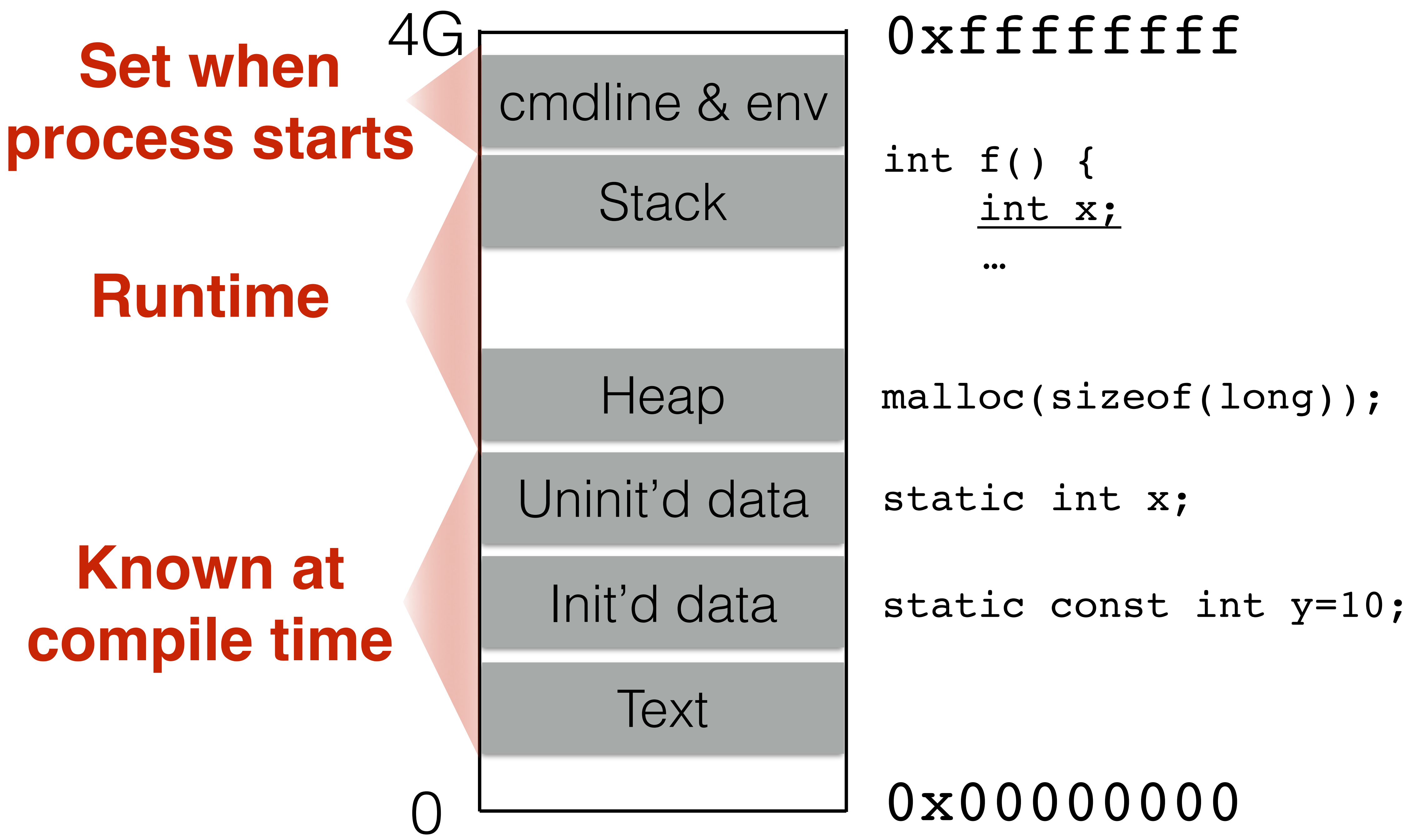
Data's location depends on how it's created



Data's location depends on how it's created

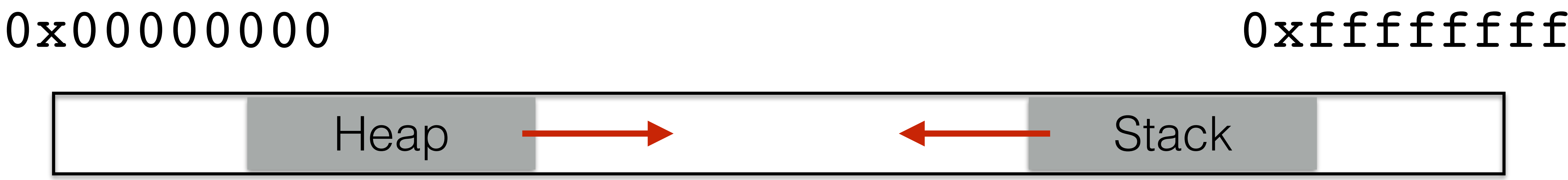


Data's location depends on how it's created



We are going to focus on runtime attacks

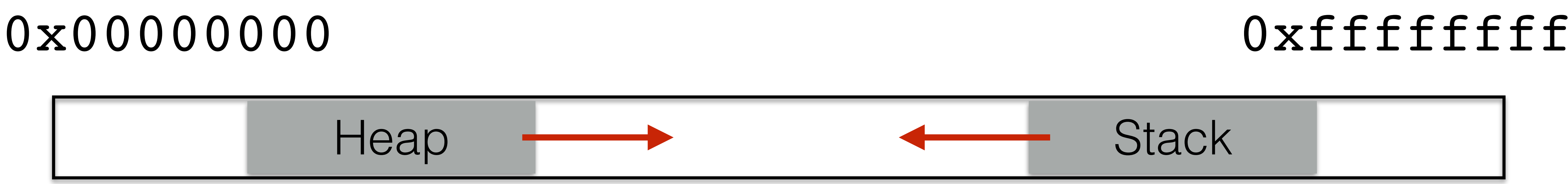
Stack and heap grow in opposite directions



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

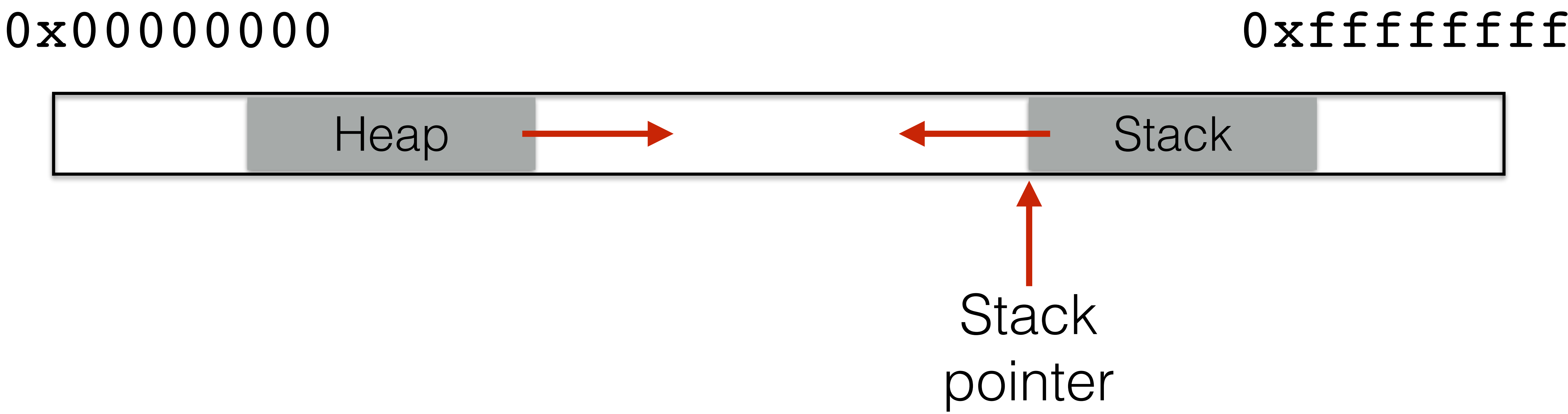
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

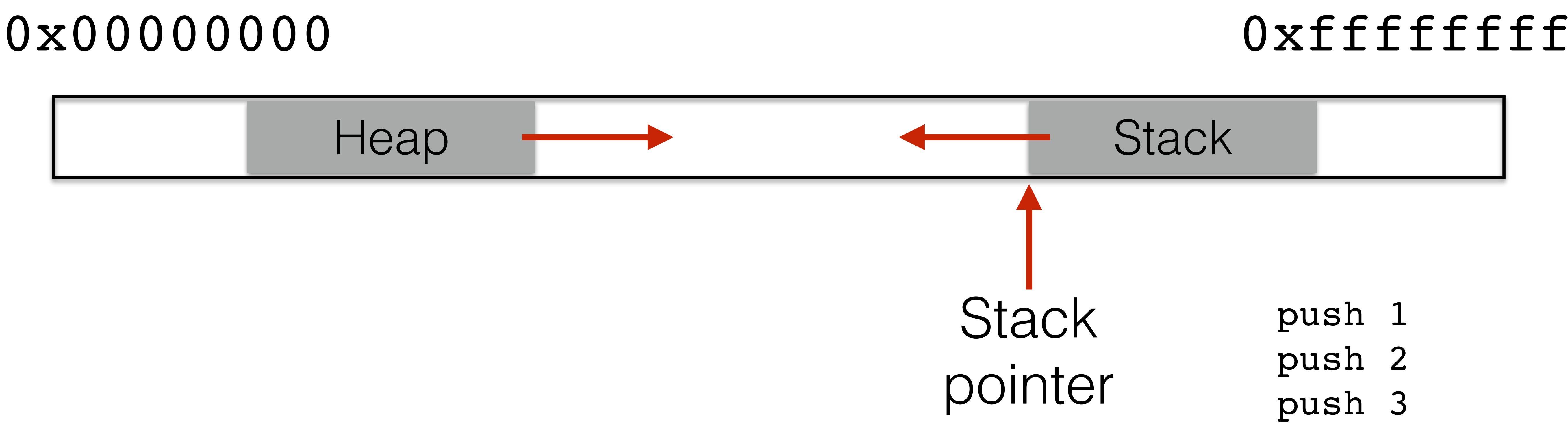
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

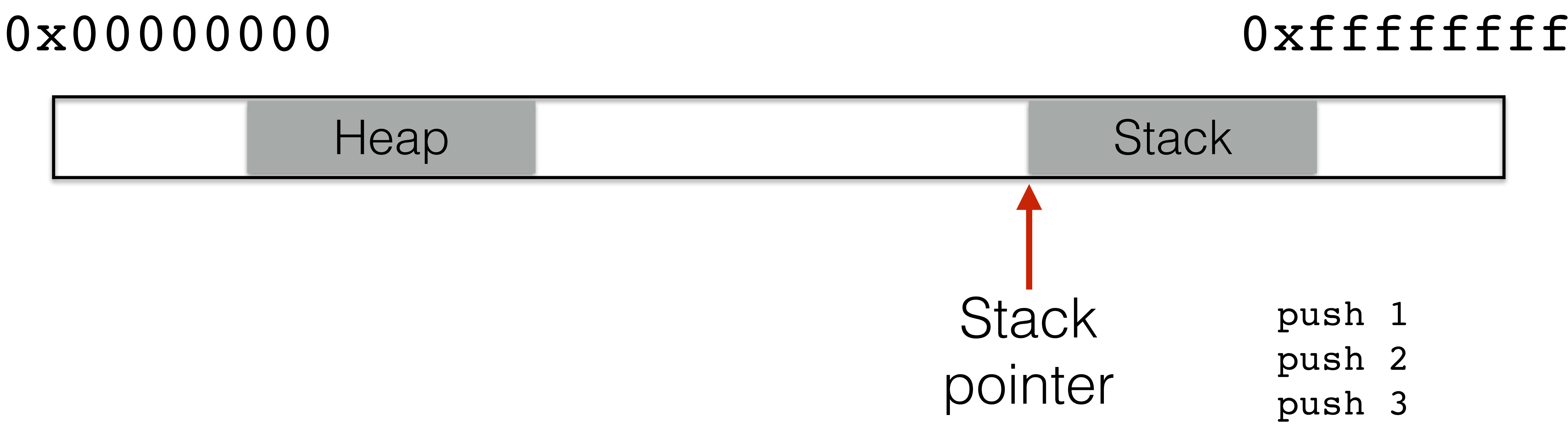
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

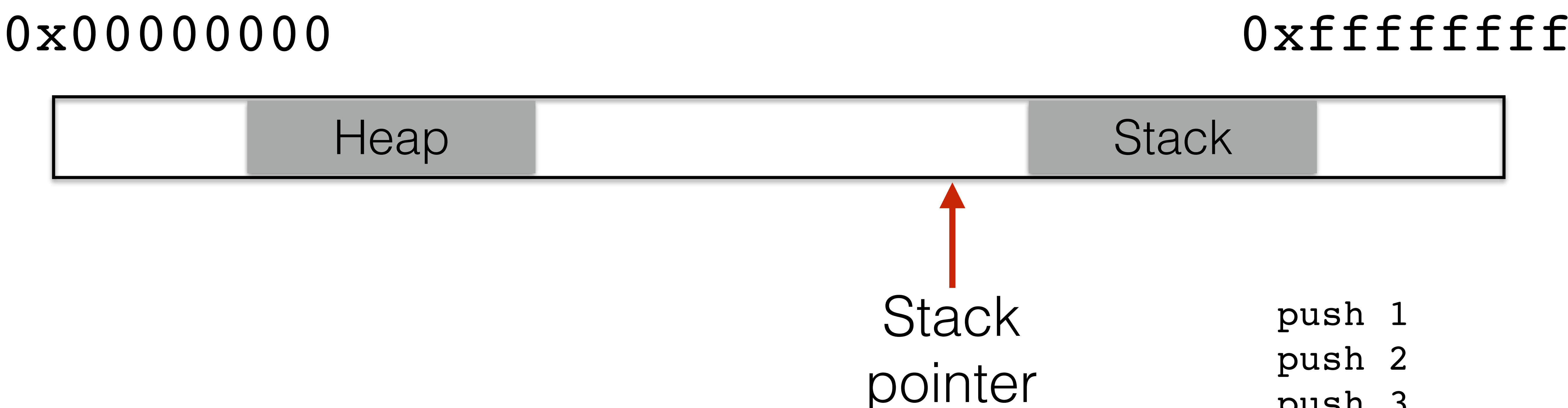
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that
adjusts the size of the stack at runtime

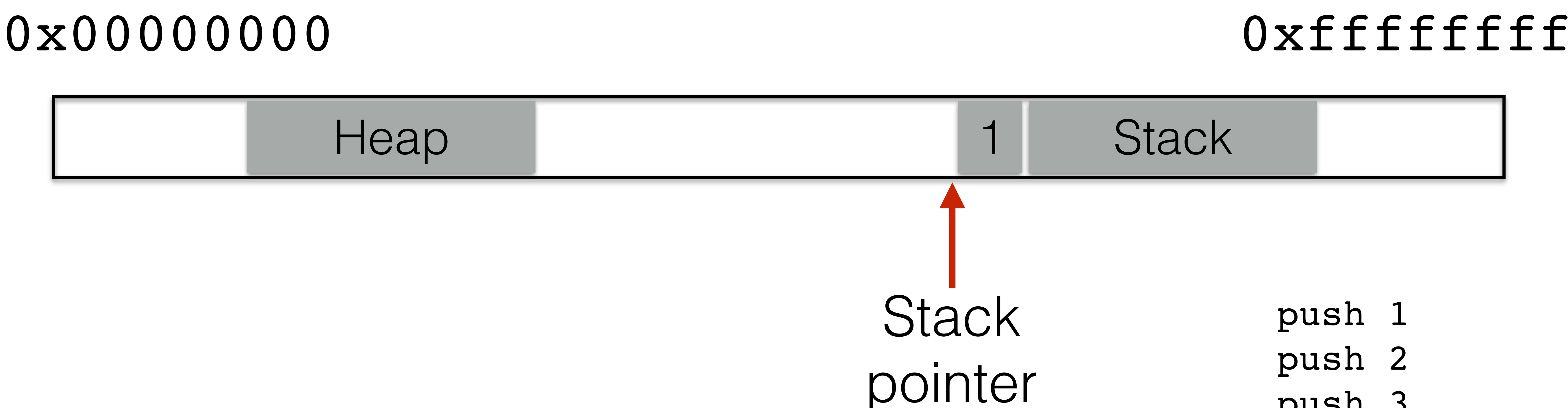



```
push 1
push 2
push 3
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

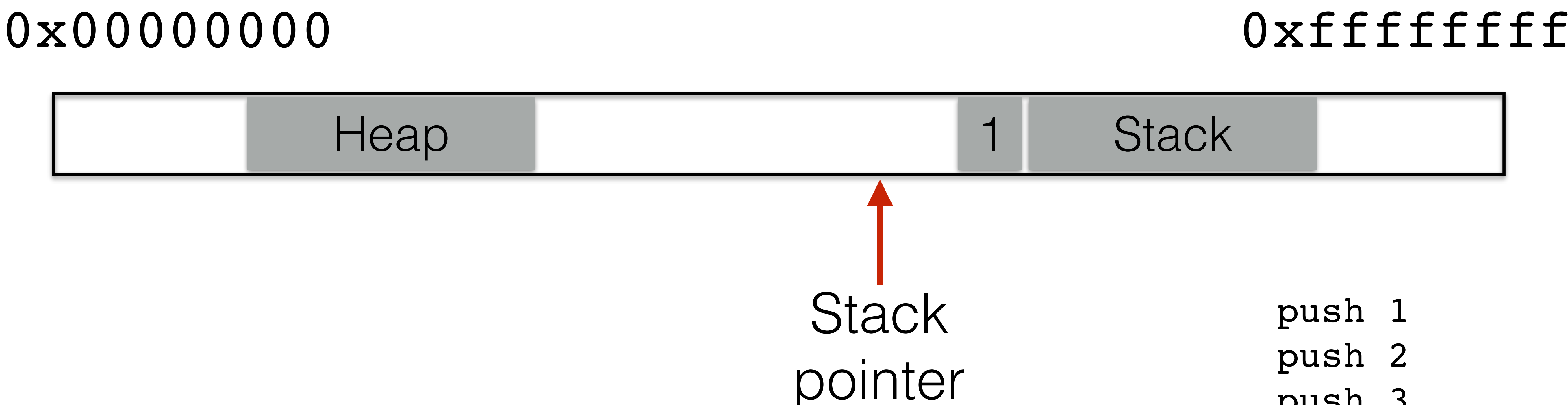



```
push 1
push 2
push 3
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

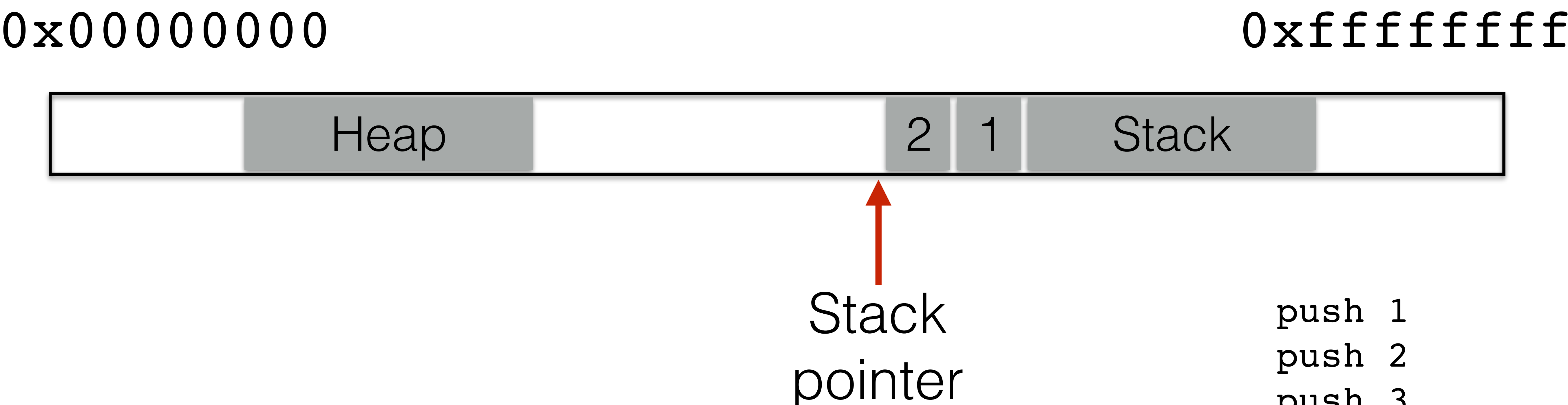



```
push 1
push 2
push 3
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that
adjusts the size of the stack at runtime

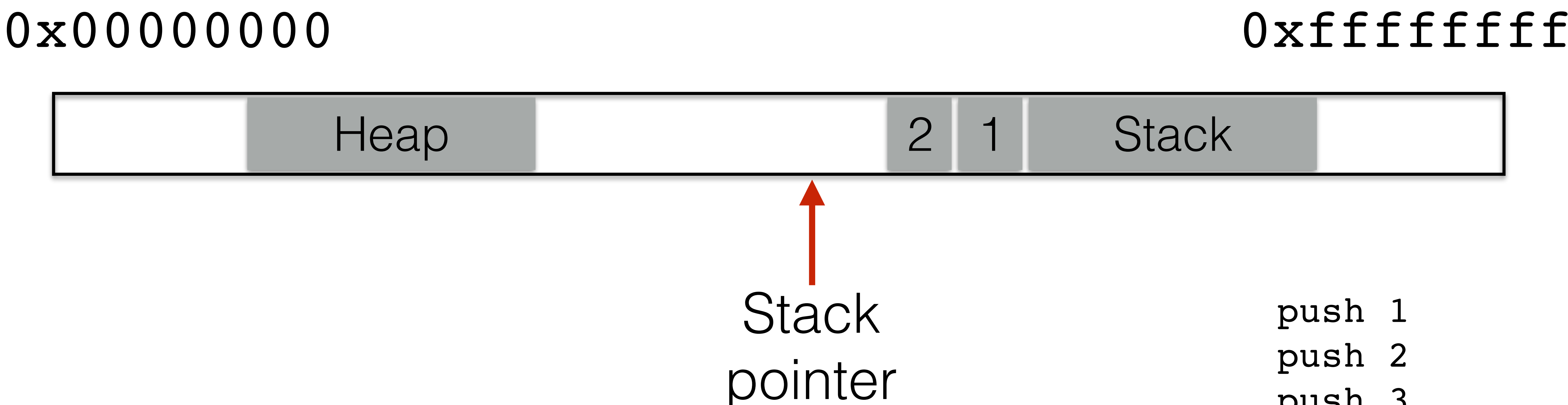



```
push 1
push 2
push 3
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

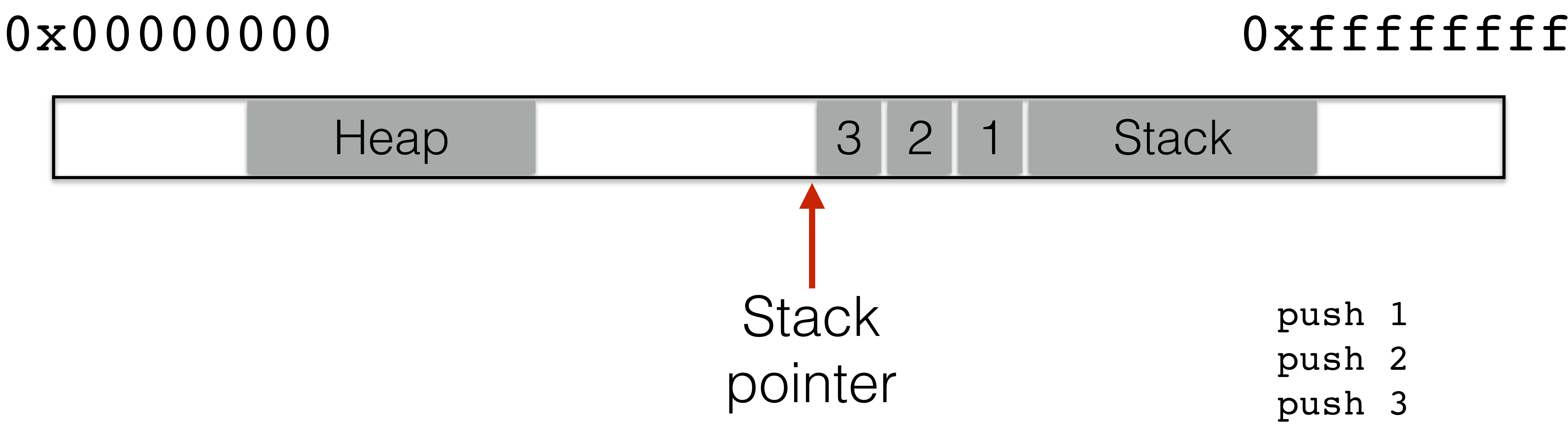



```
push 1
push 2
push 3
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

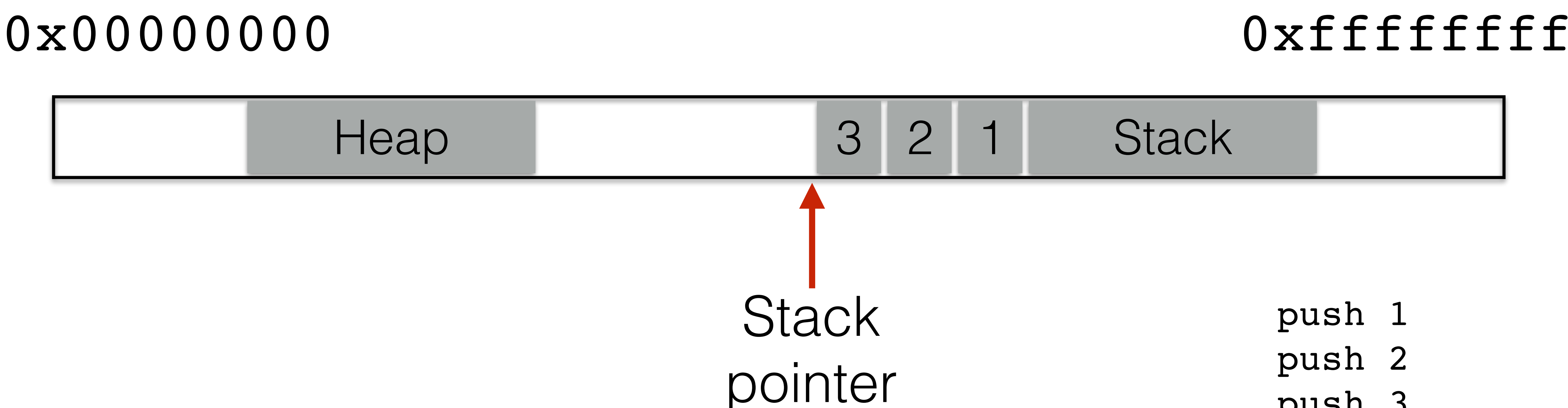
Compiler provides instructions that adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

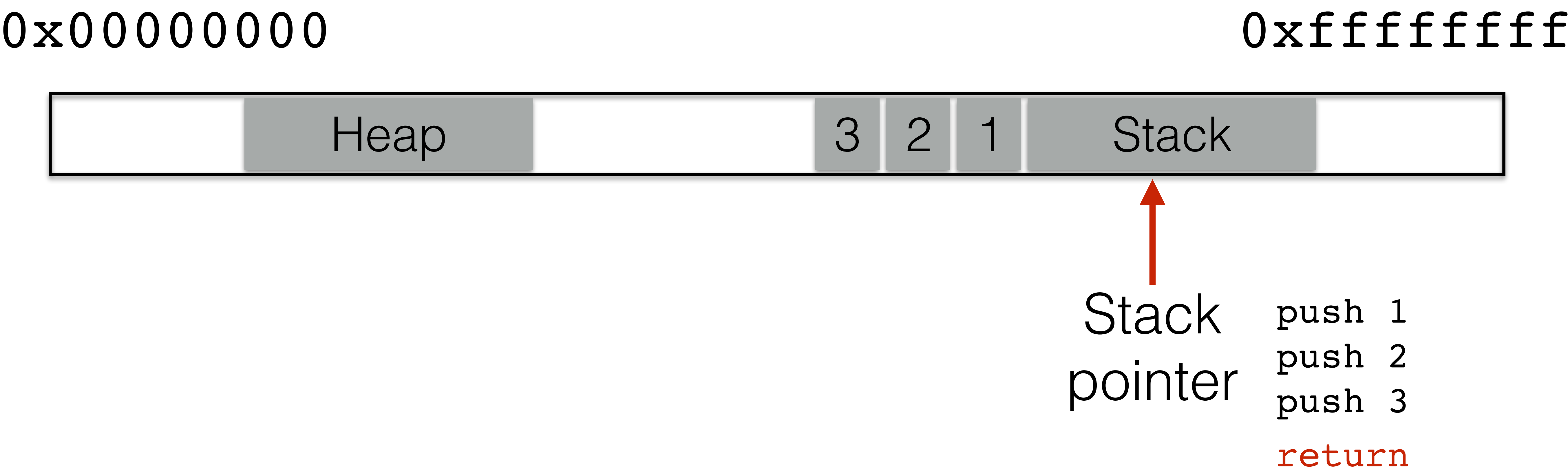



```
push 1
push 2
push 3
return
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

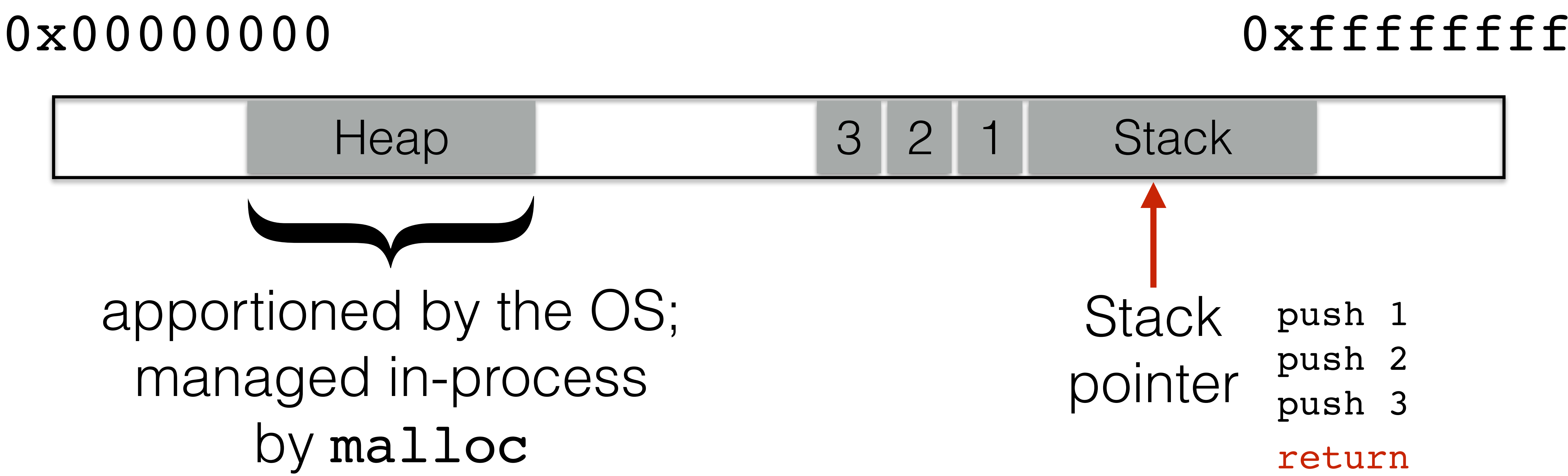
Compiler provides instructions that adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

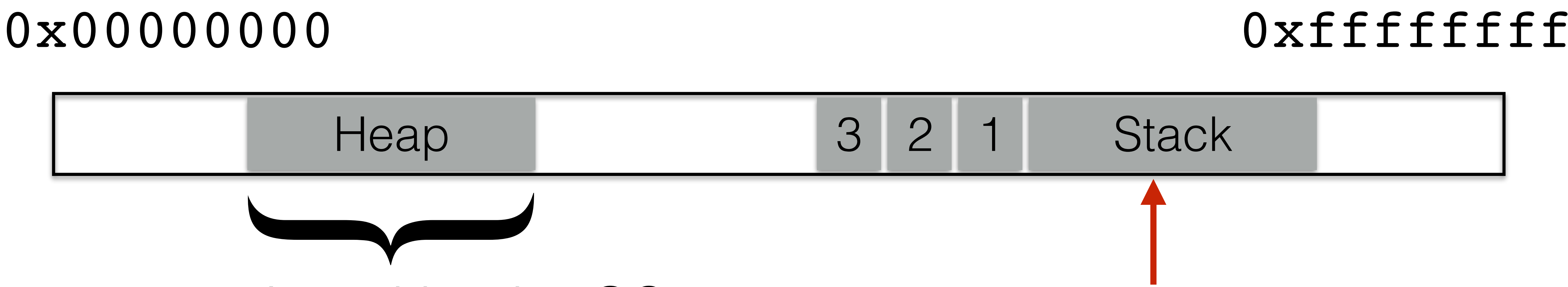
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that
adjusts the size of the stack at runtime



apportioned by the OS;
managed in-process
by `malloc`

Stack
pointer

```
push 1
push 2
push 3
return
```

Focusing on the stack for now

Stack layout when calling functions

- What do we do when we *call* a function?
    - What data need to be stored?
    - Where do they go?

- How do we *return* from a function?
    - What data need to be *restored*?
    - Where do they come from?

Code examples

(see `~/UMD/examples/` in the VM)