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

Analyzing security

“Security mindset”

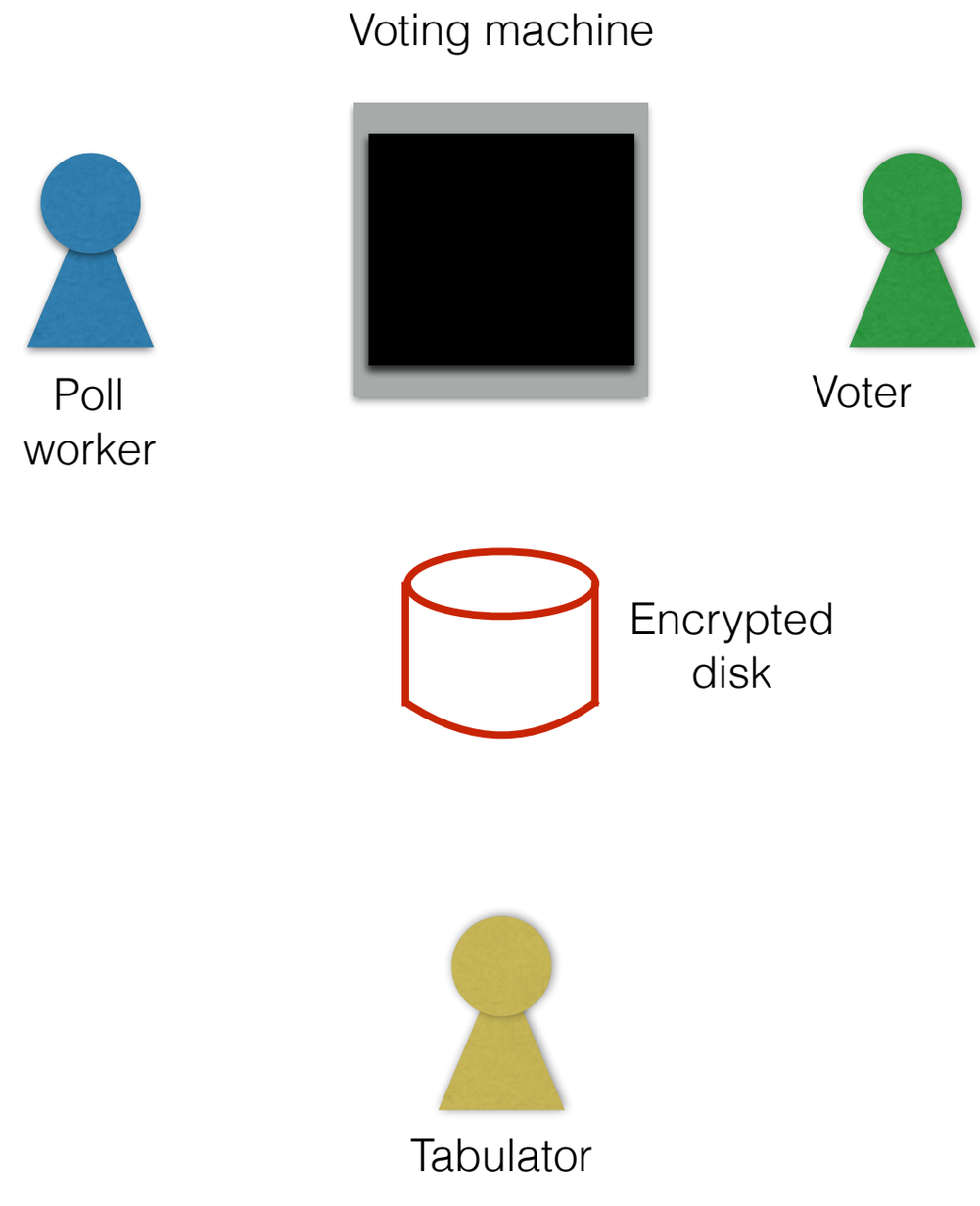
- One of the goals for this course is to develop a “security mindset”

- That is, the ability to view a potentially large, complex system and be able to reason about:
  - What are some of the **potential security threats**?
  - What are the **hidden assumptions**? (and are the explicit assumptions likely to be true?)
  - How can we **mitigate the risks** of the system?

- Be creative! (attackers will be)

E-voting analysis

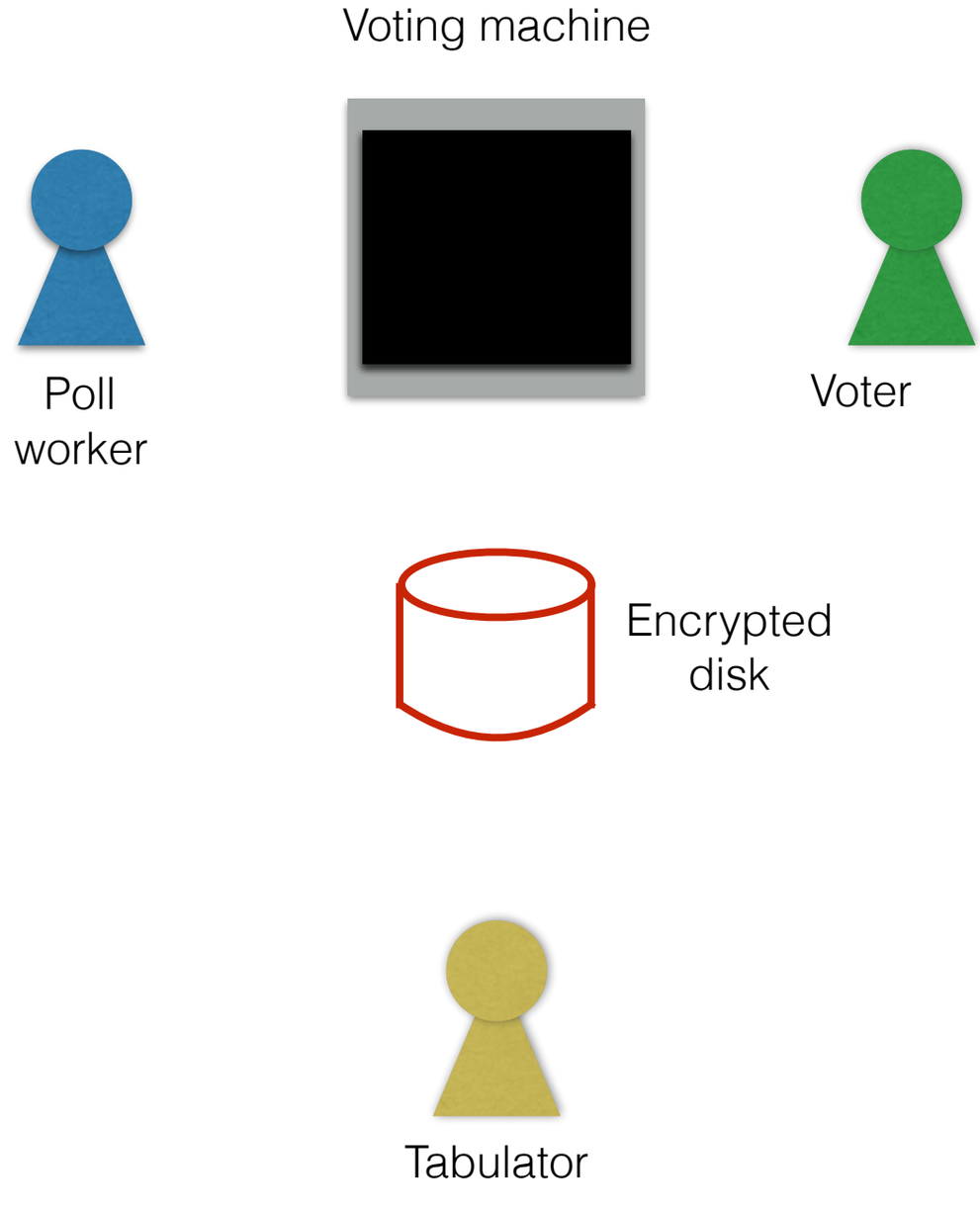
1. Summarize the system as clearly and concisely as possible



E-voting analysis

1. Summarize the system as clearly and concisely as possible

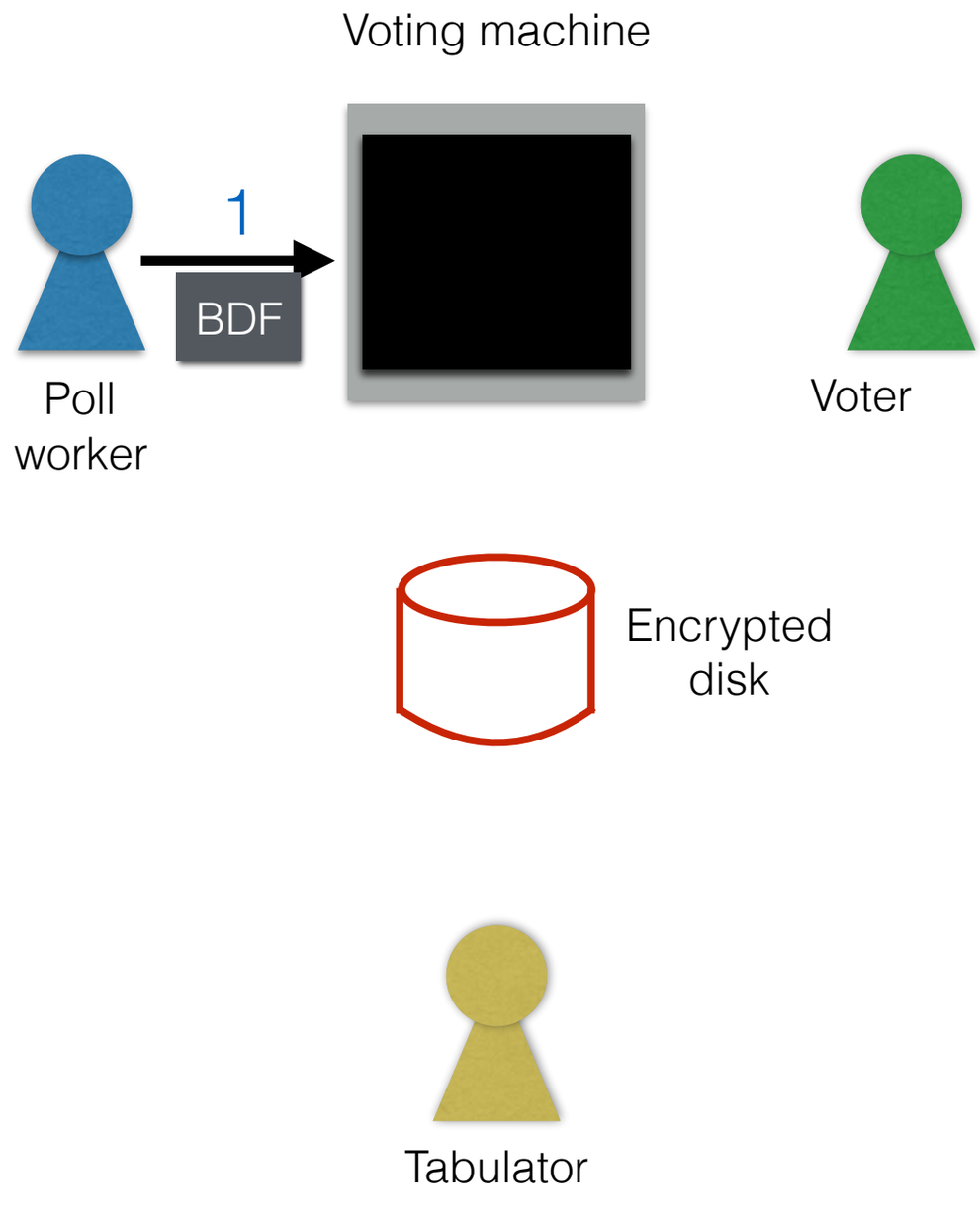
1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB



E-voting analysis

1. Summarize the system as clearly and concisely as possible

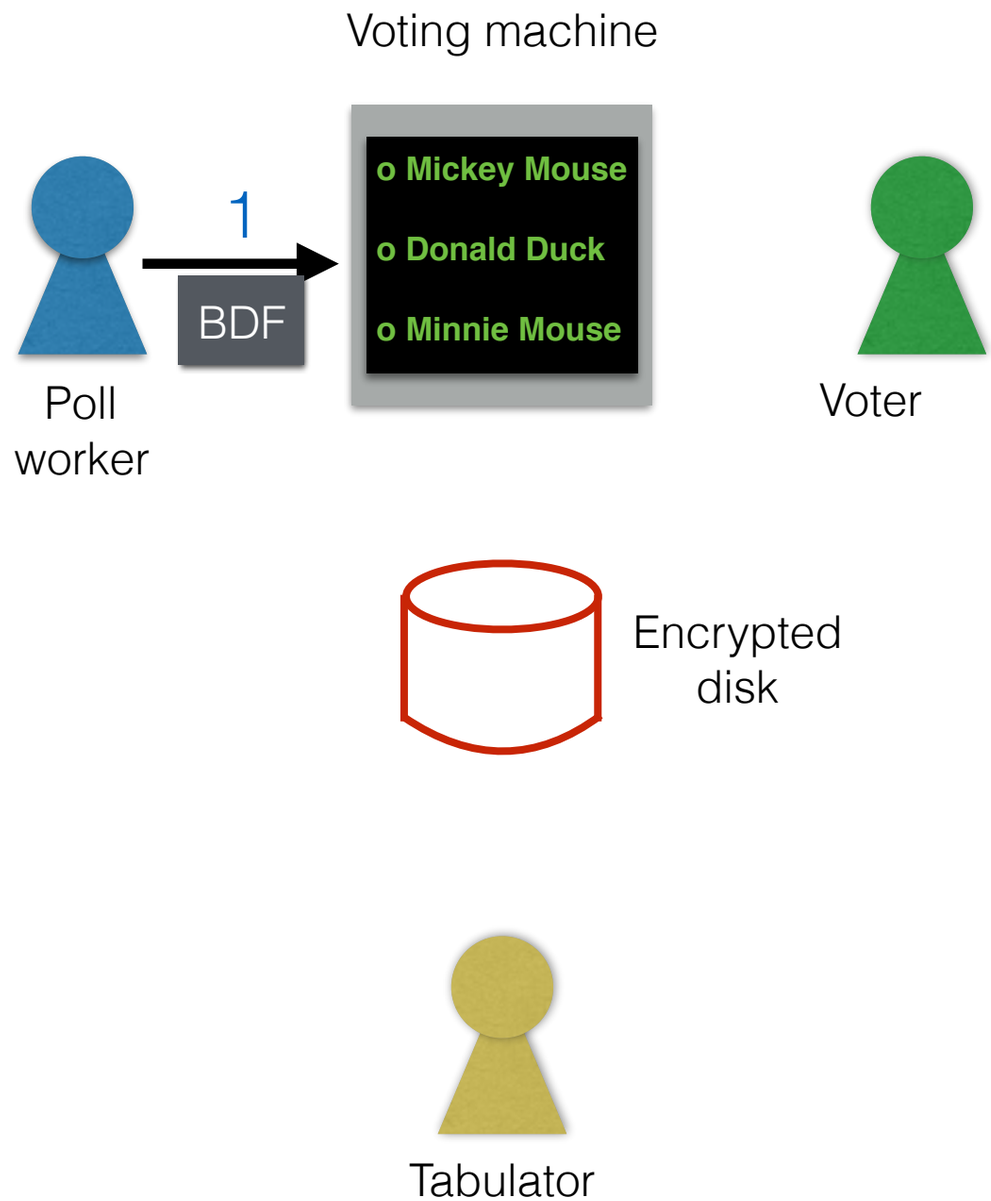
1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB



E-voting analysis

1. Summarize the system as clearly and concisely as possible

1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB

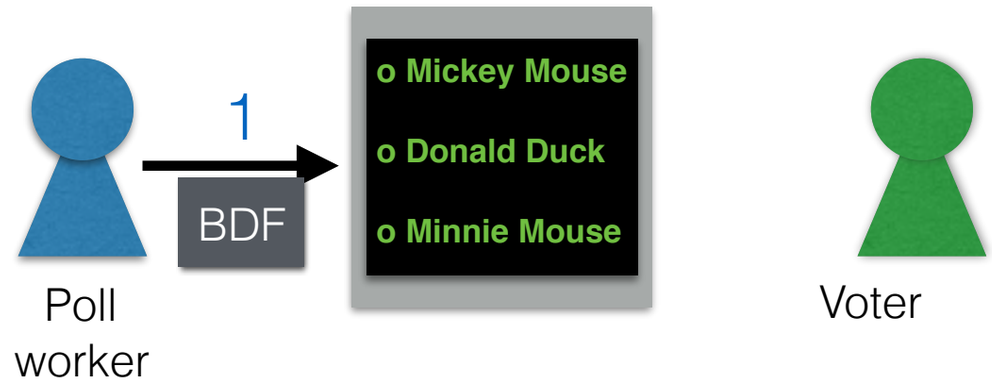


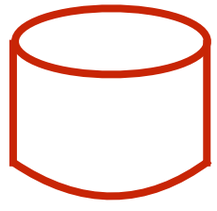
E-voting analysis

1. Summarize the system as clearly and concisely as possible

1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB

2. Voting phase
   (a) Voter obtains a single-use token from poll workers (on smartcard)
   (b) Voter uses the token to interactively vote
   (c) Vote stored encrypted on disk
   (d) Voter token canceled



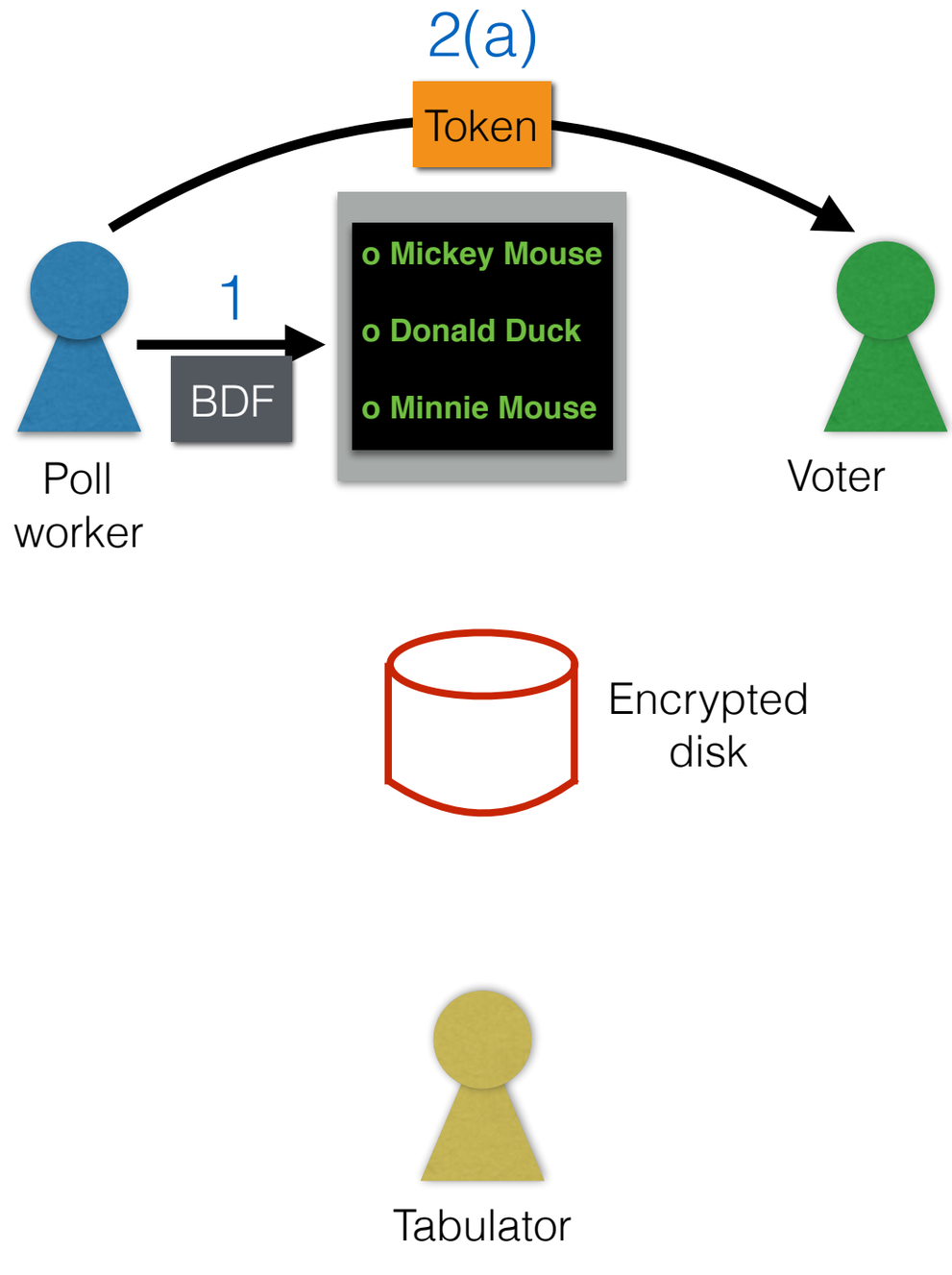


E-voting analysis

1. Summarize the system as clearly and concisely as possible

1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB

2. Voting phase
   (a) Voter obtains a single-use token from poll workers (on smartcard)
   (b) Voter uses the token to interactively vote
   (c) Vote stored encrypted on disk
   (d) Voter token canceled

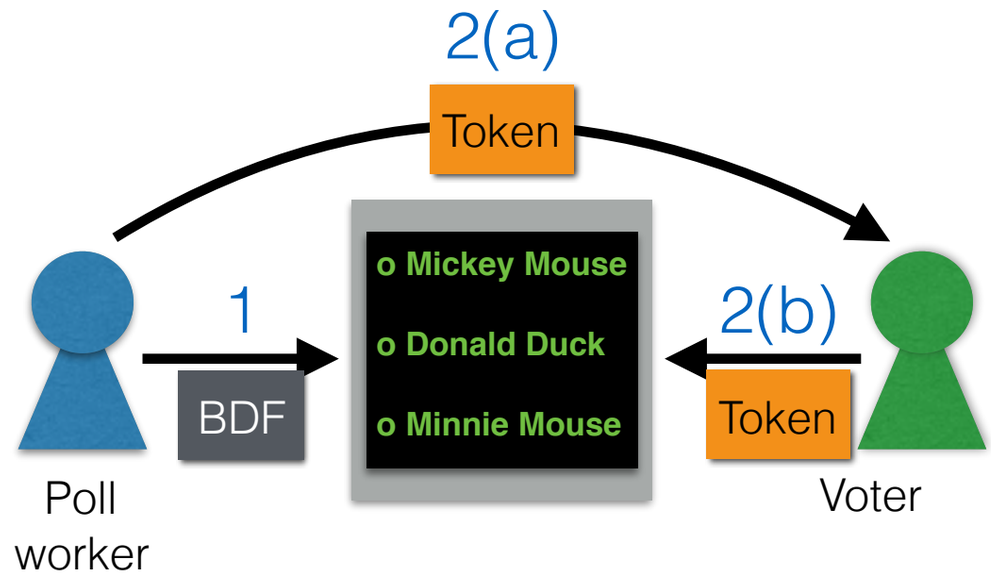


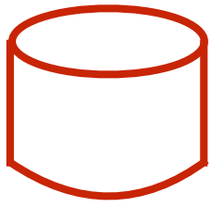
E-voting analysis

1. Summarize the system as clearly and concisely as possible

1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB

2. Voting phase
   (a) Voter obtains a single-use token from poll workers (on smartcard)
   (b) Voter uses the token to interactively vote
   (c) Vote stored encrypted on disk
   (d) Voter token canceled




Encrypted disk

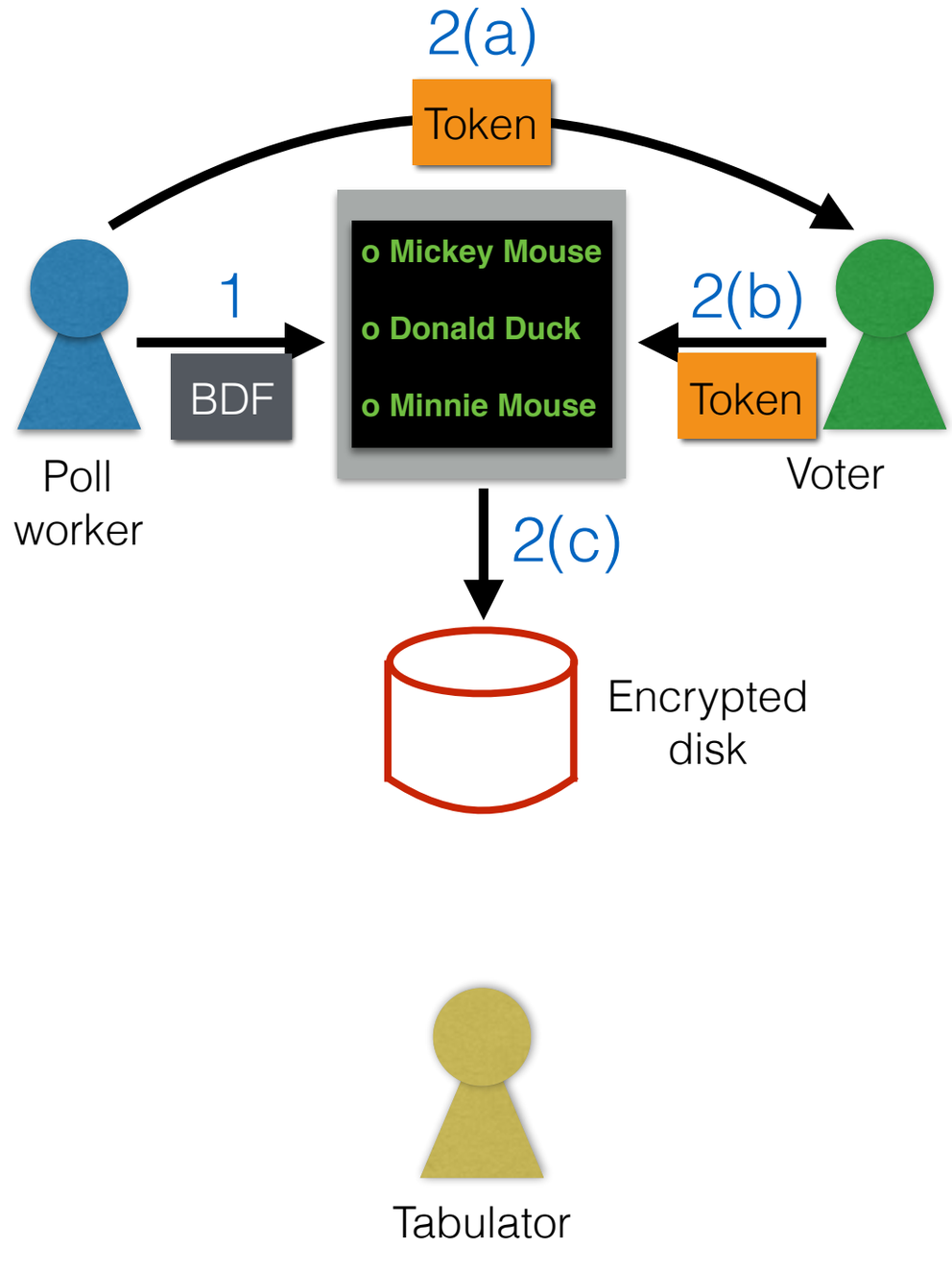

Tabulator

E-voting analysis

1. Summarize the system as clearly and concisely as possible

1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB

2. Voting phase
   (a) Voter obtains a single-use token from poll workers (on smartcard)
   (b) Voter uses the token to interactively vote
   (c) Vote stored encrypted on disk
   (d) Voter token canceled



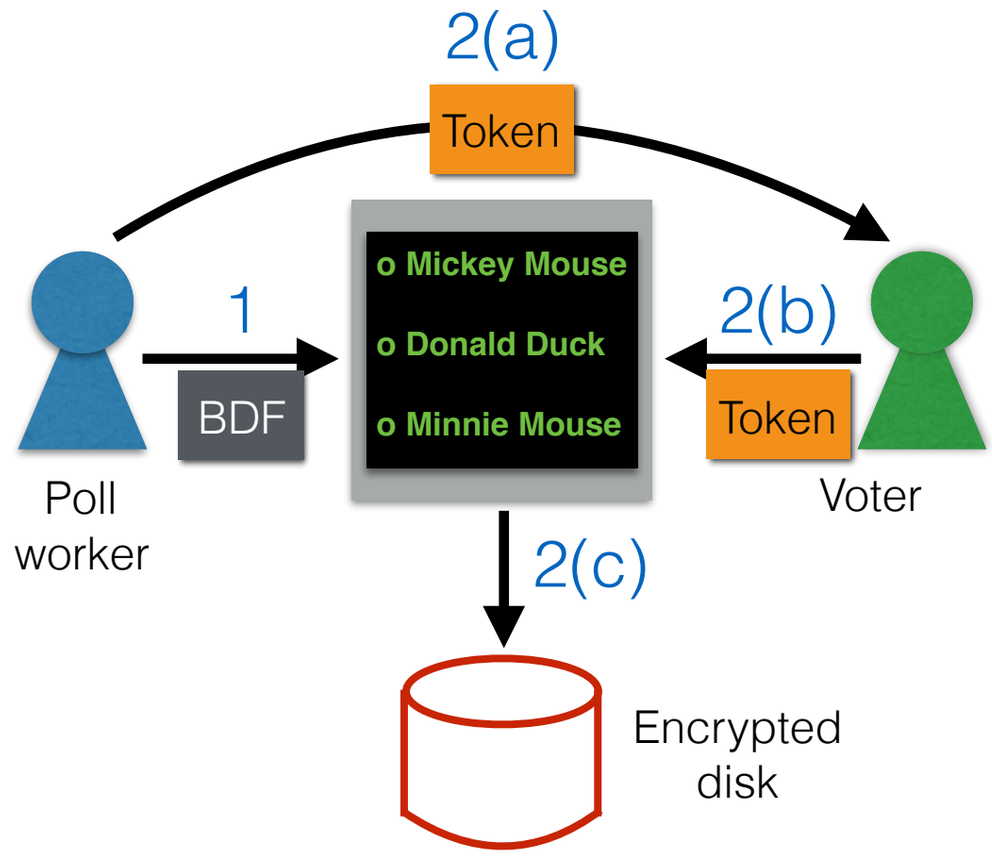
E-voting analysis

1. Summarize the system as clearly and concisely as possible

1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB

2. Voting phase
   (a) Voter obtains a single-use token from poll workers (on smartcard)
   (b) Voter uses the token to interactively vote
   (c) Vote stored encrypted on disk
   (d) Voter token canceled

3. Post-election phase
   - Stored votes decrypted and transported to tabulator
   - Tabulator counts and announces vote




Tabulator

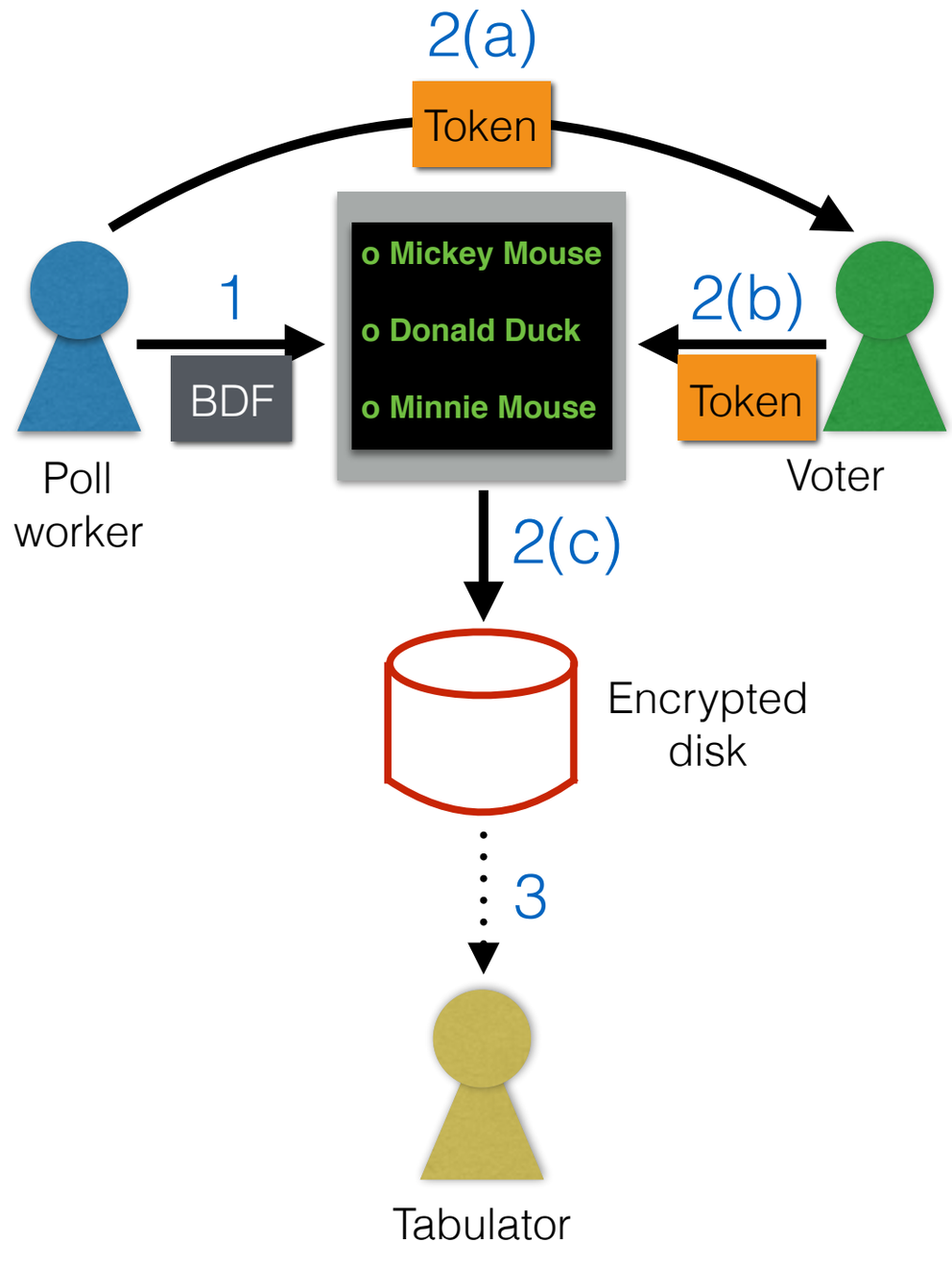
E-voting analysis

1. Summarize the system as clearly and concisely as possible

1. Pre-election phase
   - Poll worker loads a “ballot definition file” (defines who’s running, colors on the screen, and many more things) on the voting machines with, e.g., USB

2. Voting phase
   (a) Voter obtains a single-use token from poll workers (on smartcard)
   (b) Voter uses the token to interactively vote
   (c) Vote stored encrypted on disk
   (d) Voter token canceled

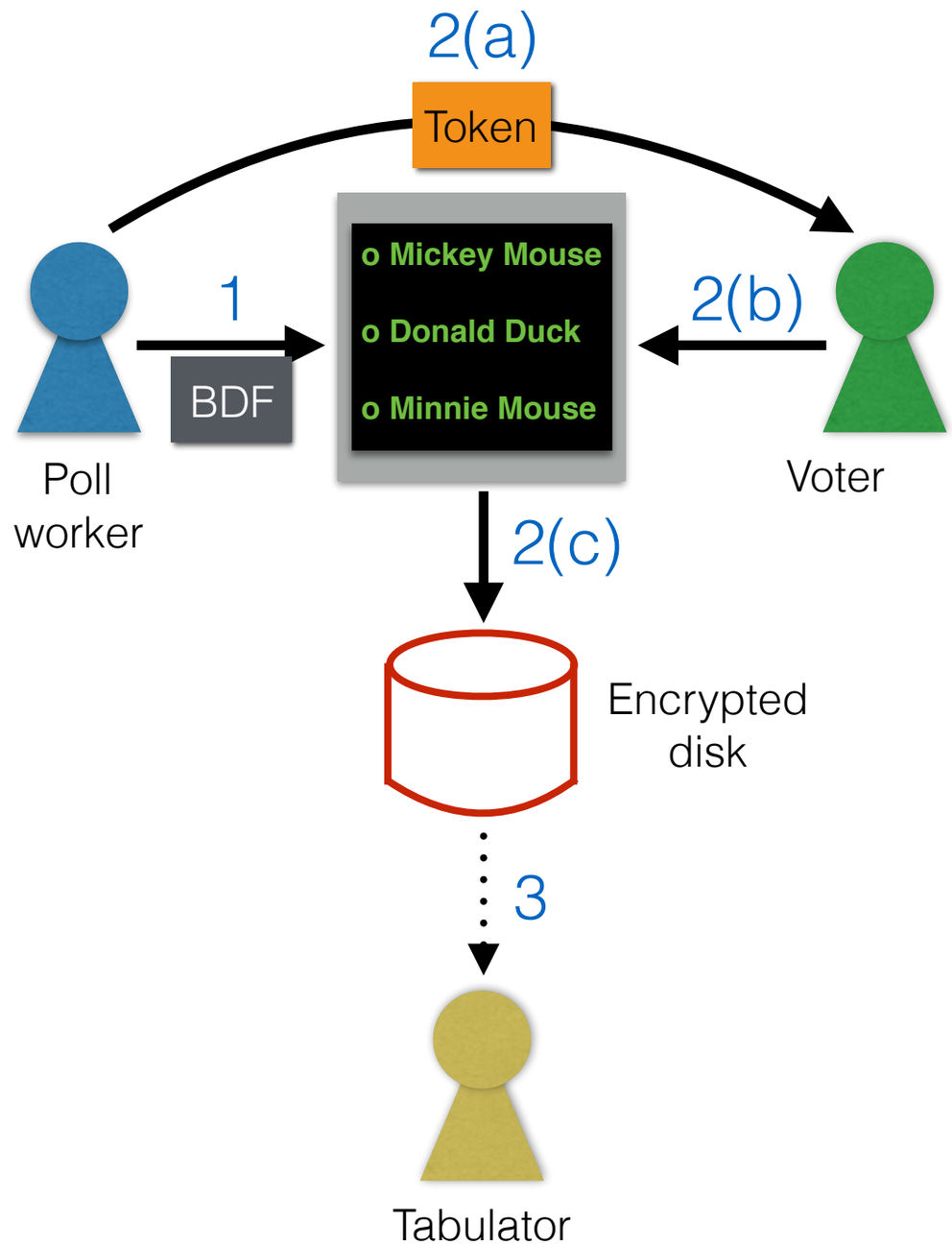
3. Post-election phase
   - Stored votes decrypted and transported to tabulator
   - Tabulator counts and announces vote



E-voting analysis

2. Identify the assets / goals of the system

- Confidentiality (can't steal data)
- Integrity (can't modify data)
- Availability (system stays up)
- Usability (no undue burden on users)

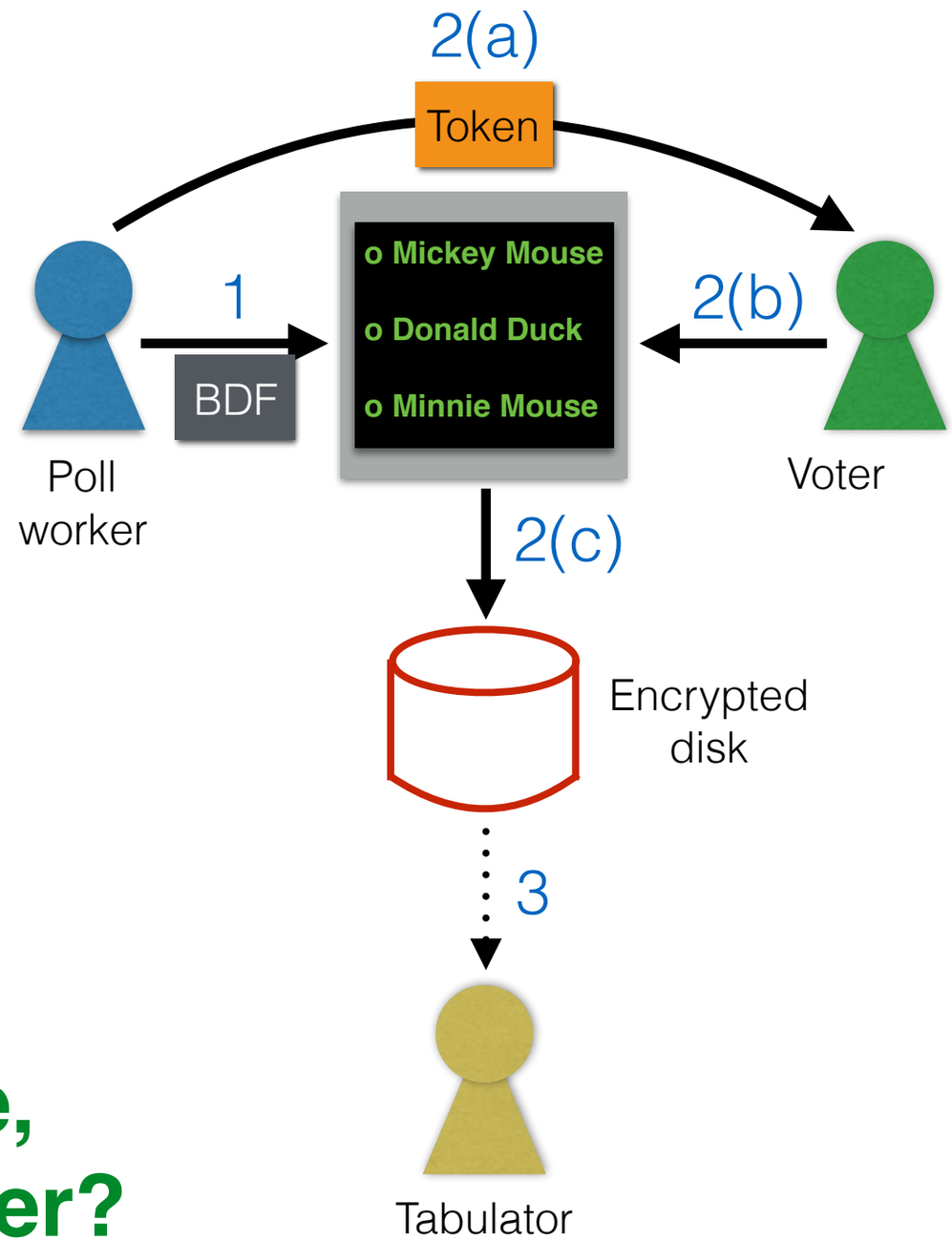


E-voting analysis

2. Identify the assets / goals of the system

- Confidentiality (can't steal data)
- Integrity (can't modify data)
- Availability (system stays up)
- Usability (no undue burden on users)

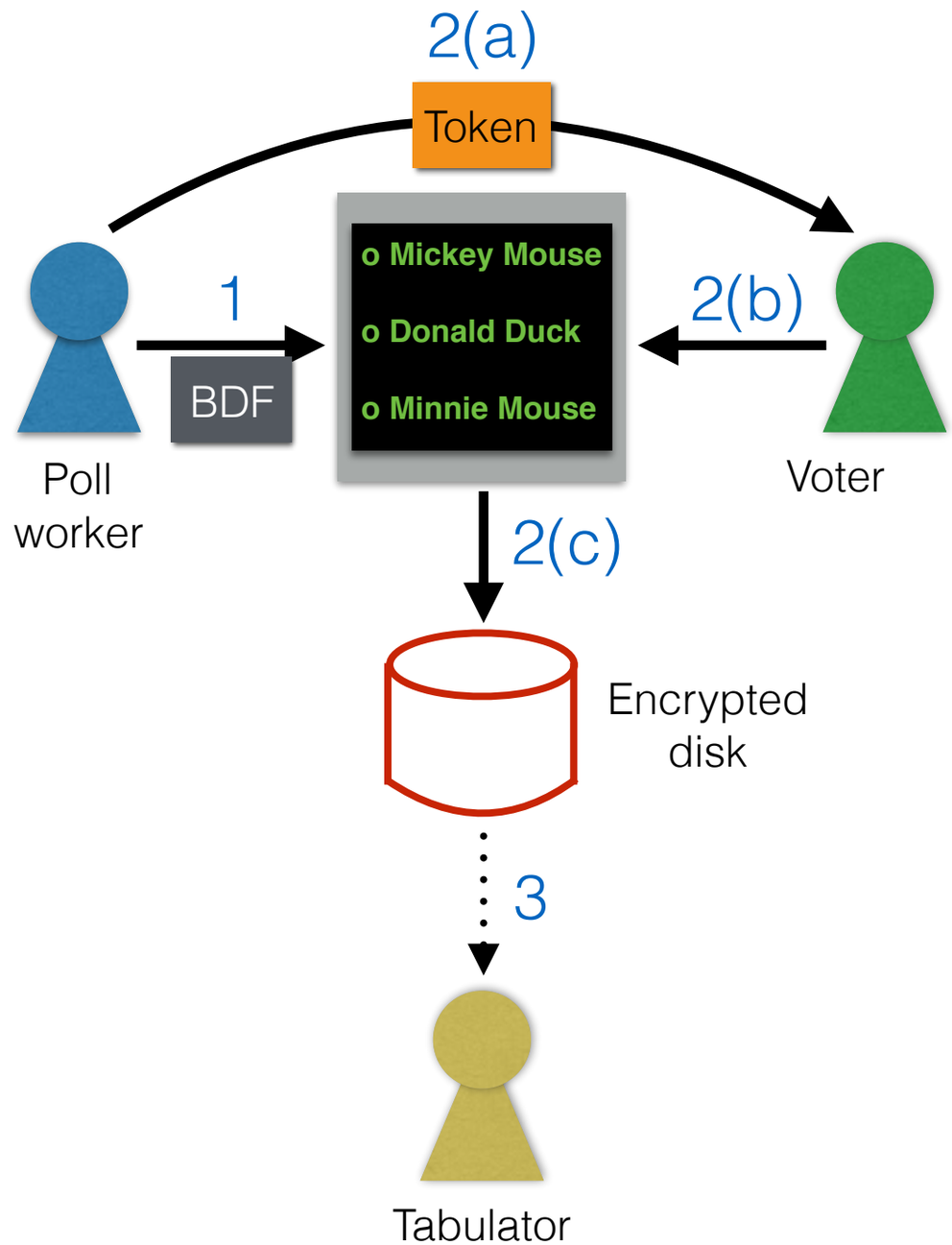
Is it ok if the attacker *can* do these, so long as you can *catch* him or her?



E-voting analysis

2. Identify the assets / goals of the system

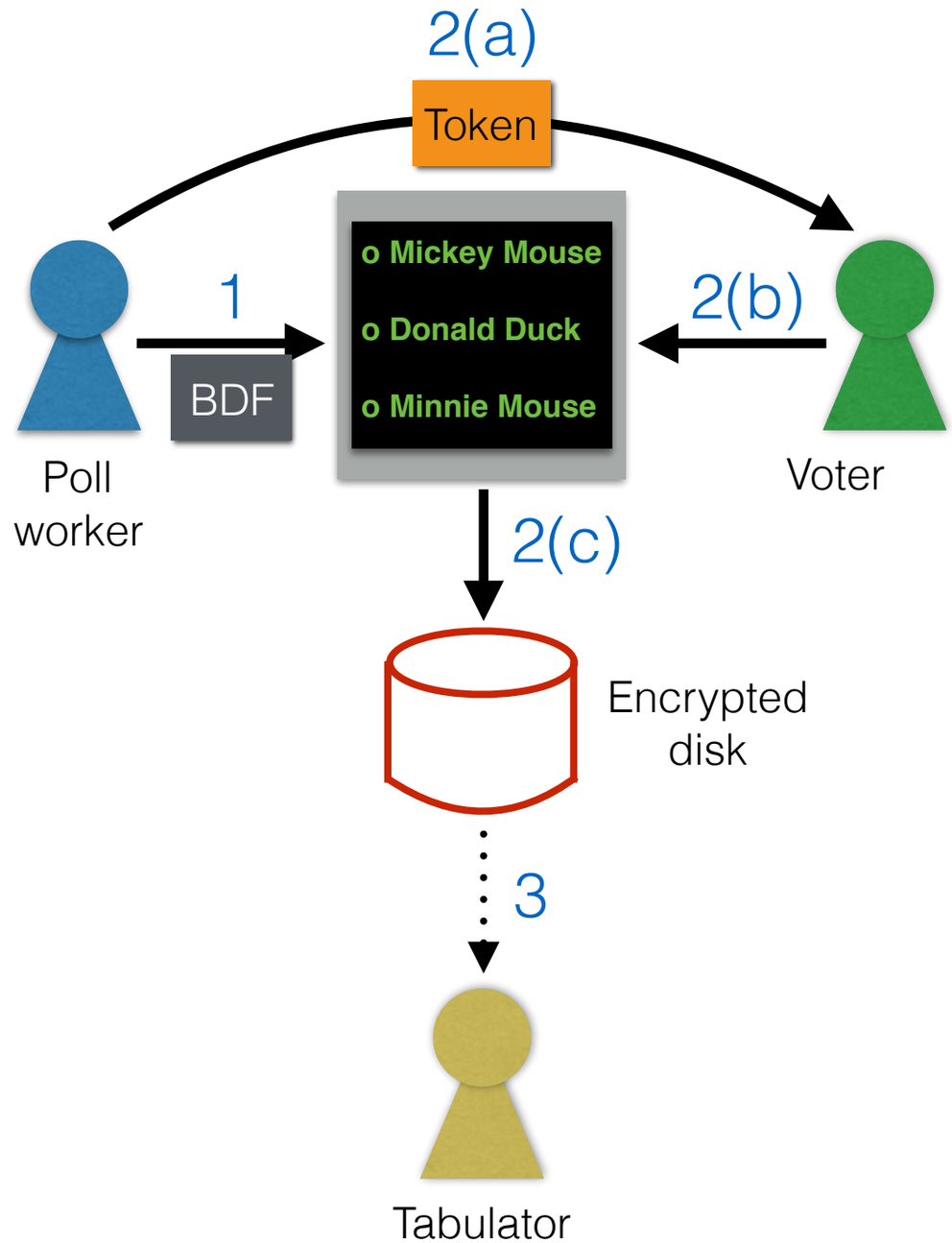
- Confidentiality (can't steal data)
- Integrity (can't modify data)
- Availability (system stays up)
- Usability (no undue burden on users)



E-voting analysis

2. Identify the assets / goals of the system

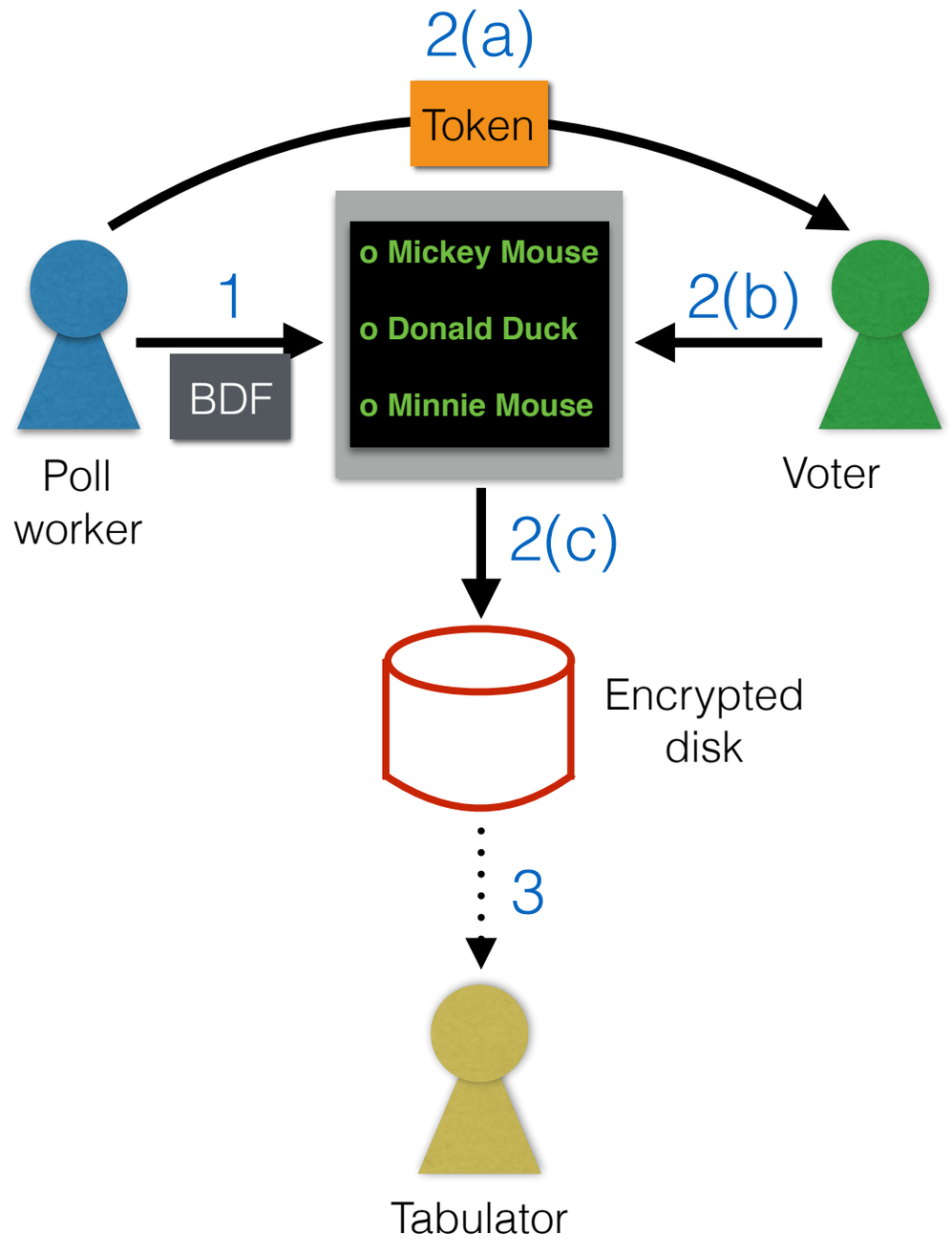
- Confidentiality (can't steal data)
  - No one knows for whom any given voter voted (except for the voter)
- Integrity (can't modify data)
- Availability (system stays up)
- Usability (no undue burden on users)



E-voting analysis

2. Identify the assets / goals of the system

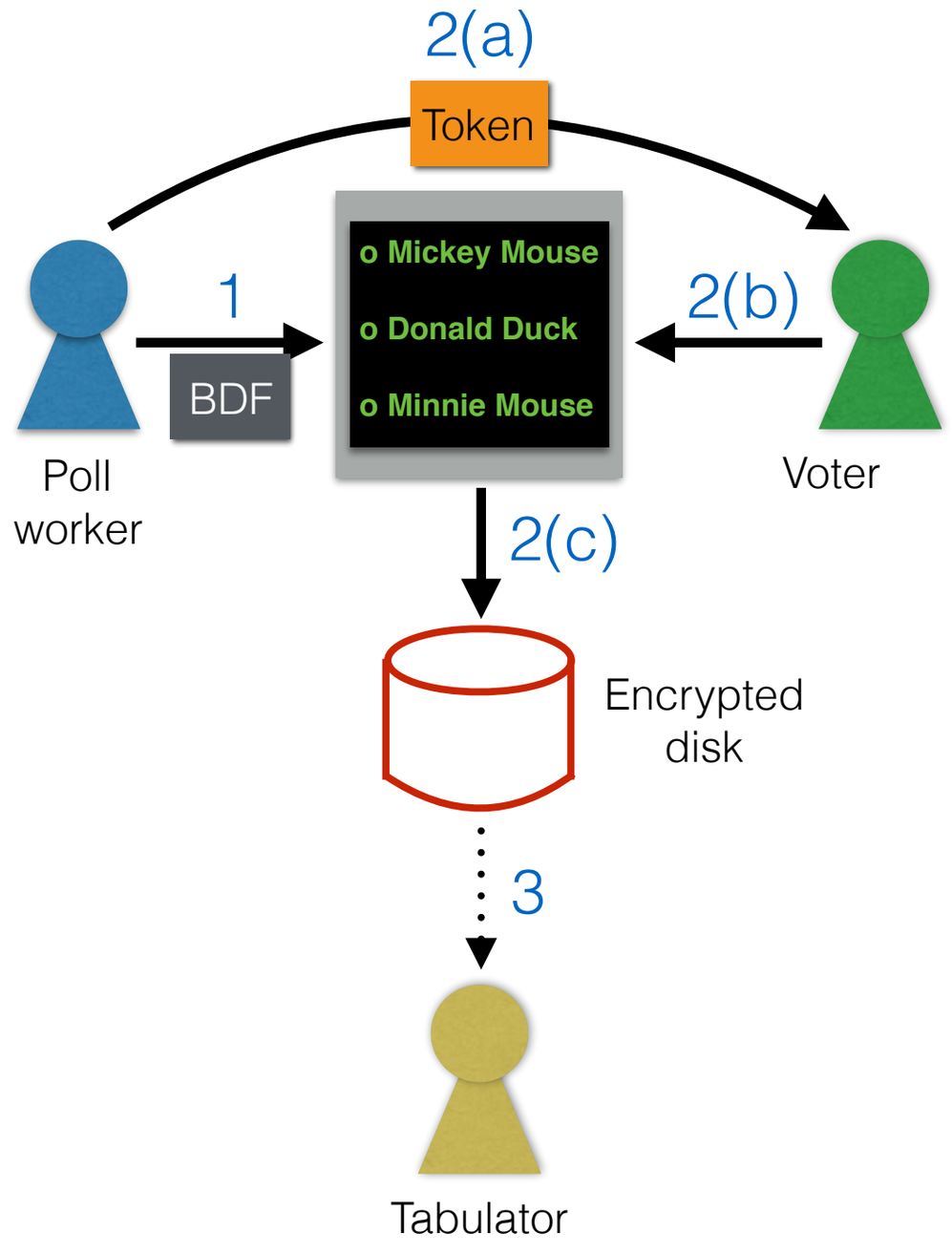
- Confidentiality (can't steal data)
  - No one knows for whom any given voter voted (except for the voter)
- Integrity (can't modify data)
  - Every voter's vote counted *once*
  - No voter's vote changed
- Availability (system stays up)
- Usability (no undue burden on users)



E-voting analysis

2. Identify the assets / goals of the system

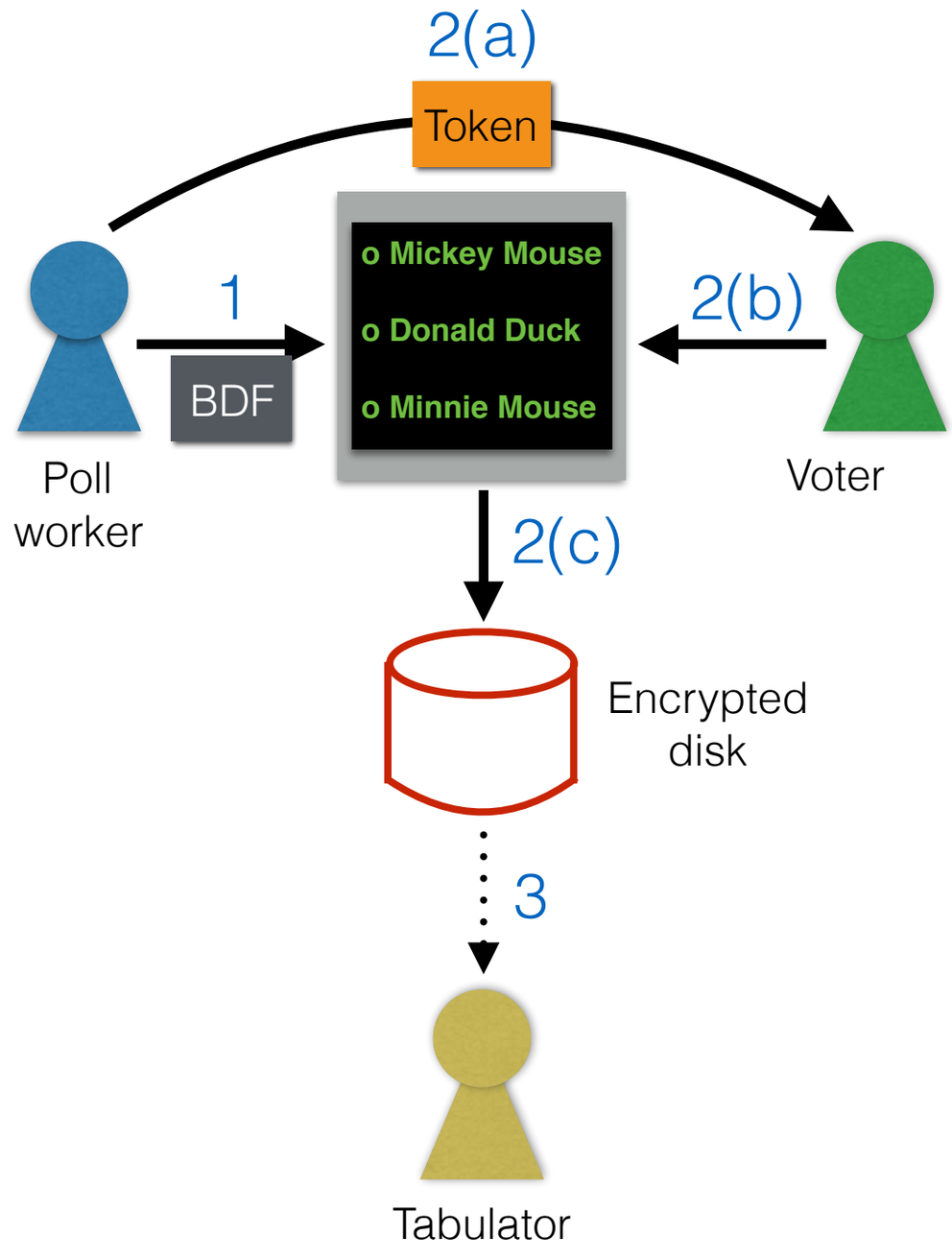
- Confidentiality (can't steal data)
  - No one knows for whom any given voter voted (except for the voter)
- Integrity (can't modify data)
  - Every voter's vote counted *once*
  - No voter's vote changed
- Availability (system stays up)
  - Everyone has the ability to cast their vote
- Usability (no undue burden on users)



E-voting analysis

2. Identify the assets / goals of the system

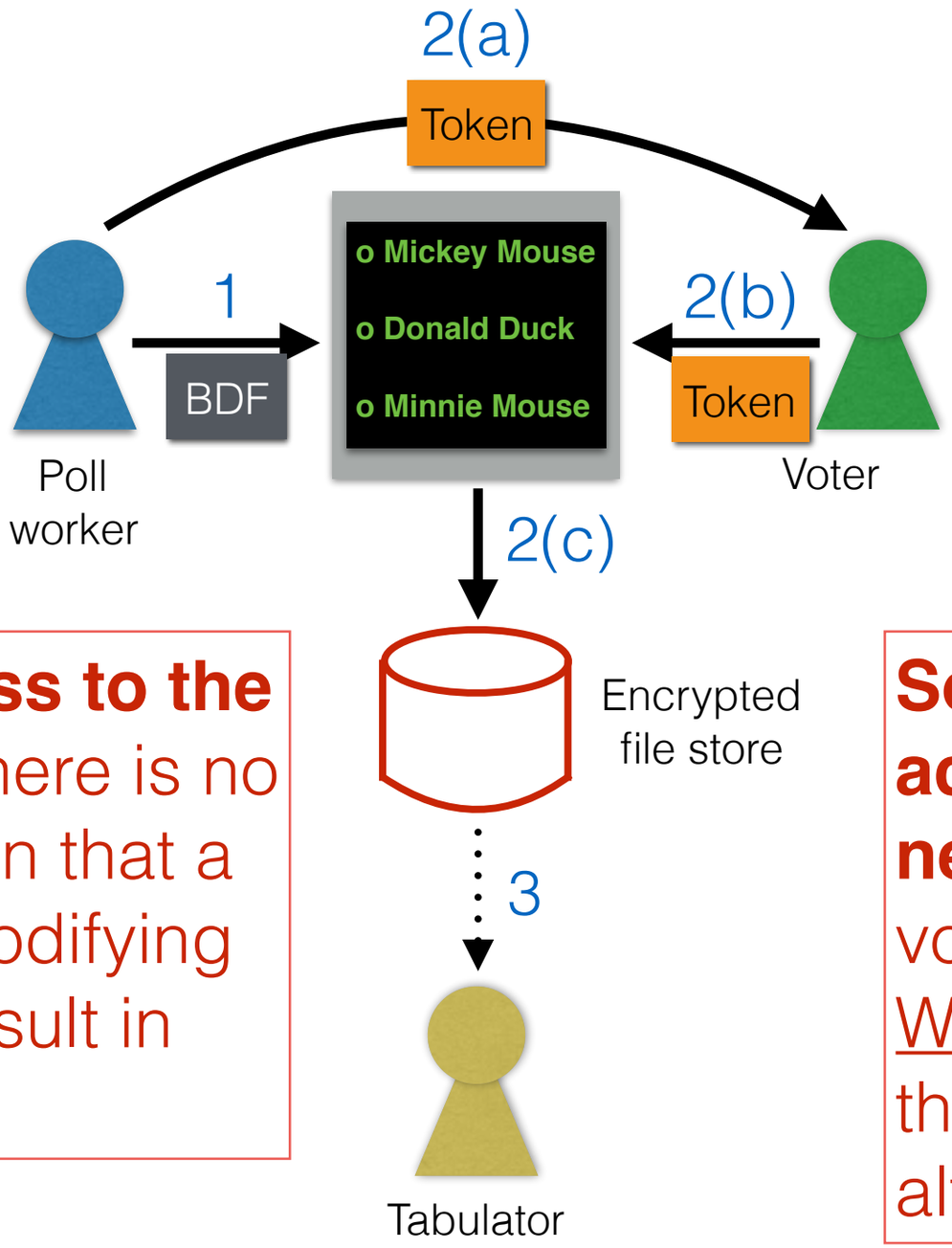
- Confidentiality (can't steal data)
  - No one knows for whom any given voter voted (except for the voter)
- Integrity (can't modify data)
  - Every voter's vote counted *once*
  - No voter's vote changed
- Availability (system stays up)
  - Everyone has the ability to cast their vote
- Usability (no undue burden on users)
  - Easy for the voter to vote (correct language, good UI)
  - Easy for the tabulator to count votes



E-voting analysis

3. Identify the adversaries and threats

Poll worker could set BDF to print "Mickey Mouse" but record as "Minnie Mouse"



Voter could attempt to generate their own tokens & get ≥2 votes

Someone with access to the machine: Because there is no end-to-end verification that a vote was counted, modifying the software could result in complete control

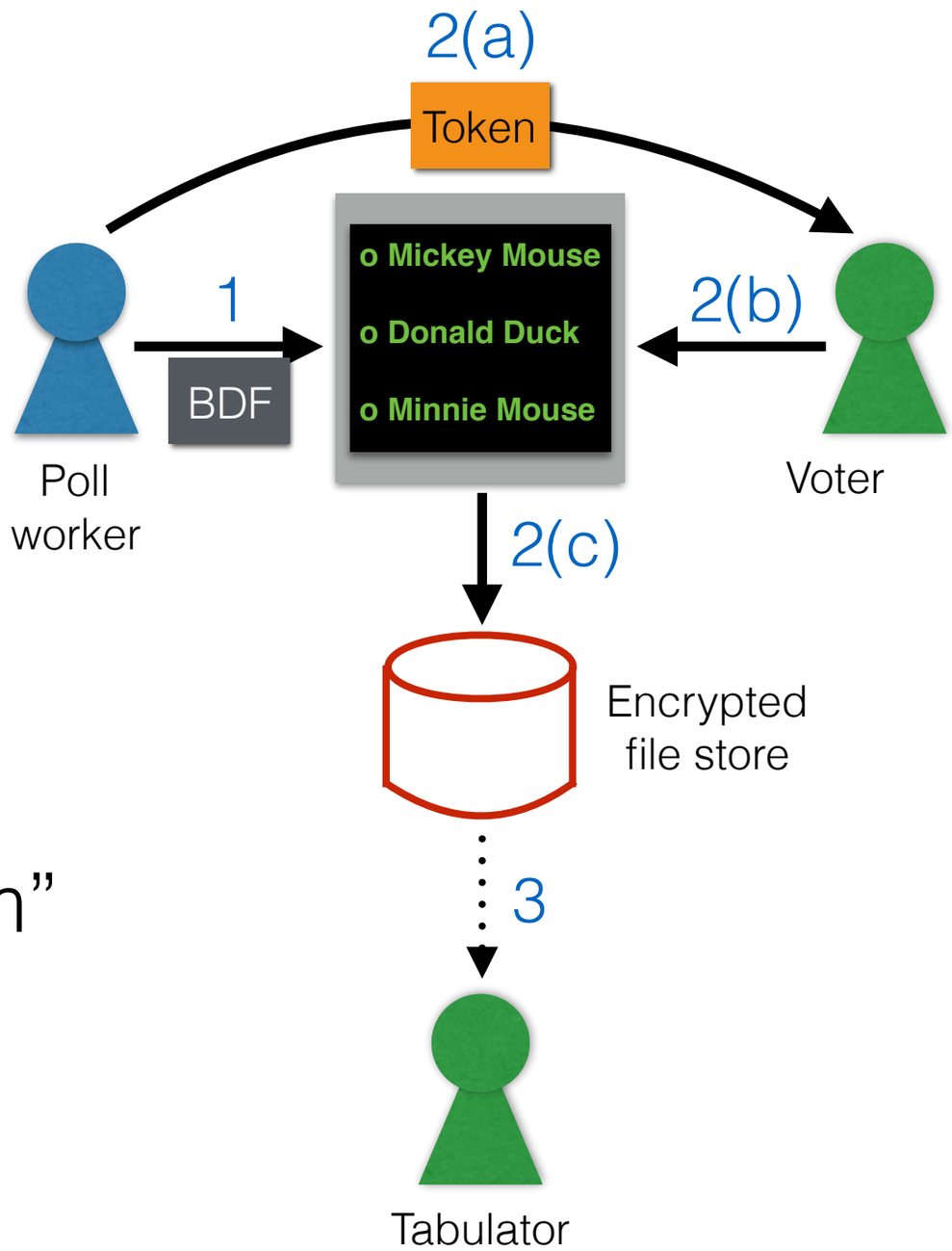
Someone with access to the disk or network: Read who voted for whom. Writing could change the outcome altogether

E-voting analysis

4. Identify the vulnerabilities

Not a theoretical system:

Diebold AccuVote-TS
Used in 37 states at time of study



Optional reading

"Analysis of an Electronic Voting System"
Kohno et al.

E-voting analysis

4. Identify the vulnerabilities

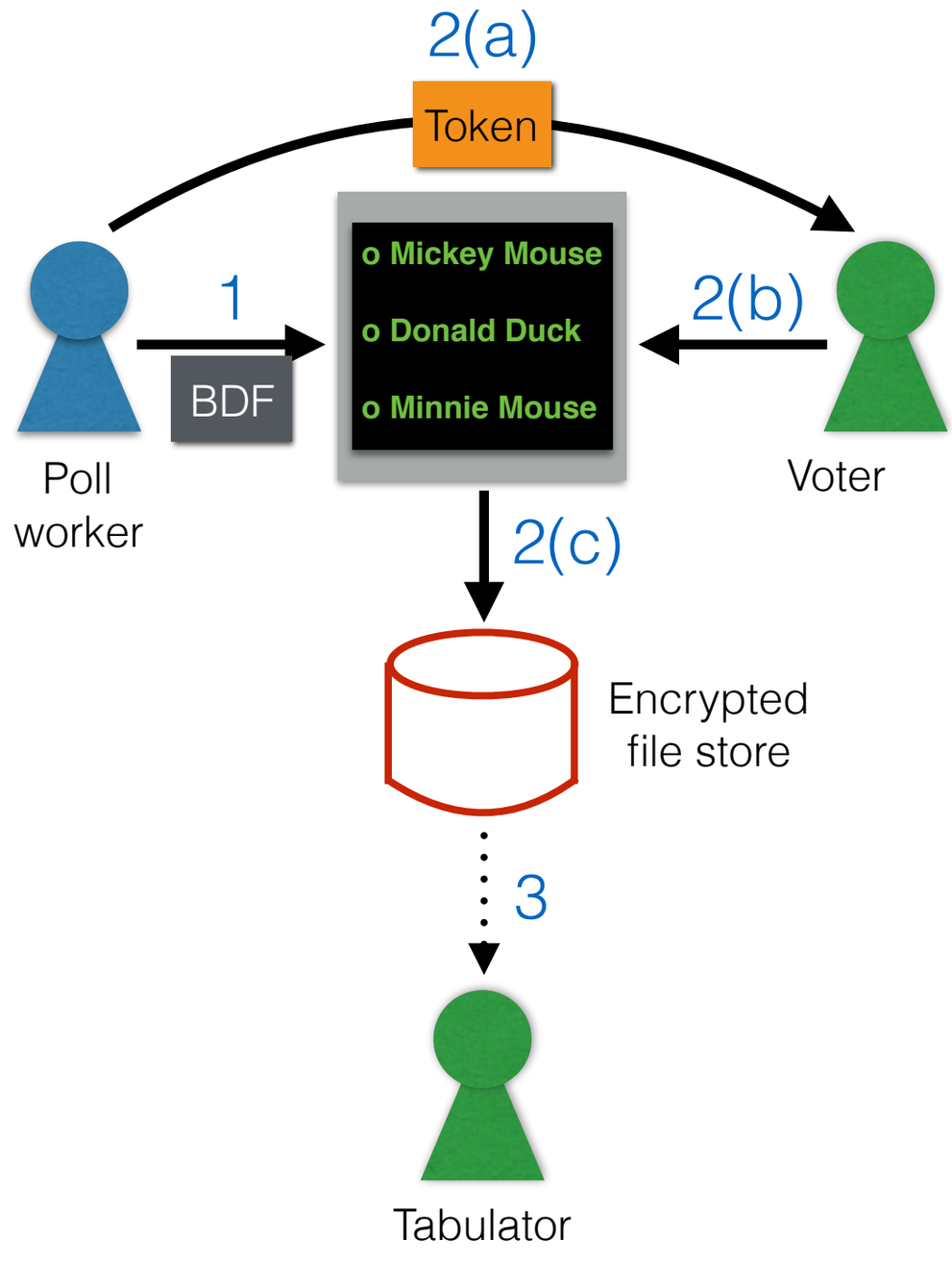
- Ballot definition files are not authenticated
  - How do we know they're from the election board?
  - Can redefine "Candidate A" as "Candidate B"
  - Viruses

- Smartcards are not authenticated
  - How do we know they're not user-generated?
  - Possible to make your own and vote multiple times.

- Specific software vulnerabilities
  - Every machine has the same encryption key!
  - Break one, and they all fall

- Votes are shipped unencrypted!

- Votes are stored in the order cast
  - If one can view the data unencrypted, this violates our confidentiality goal



Analyzing security

Takeaway points

- Analyzing security requires a whole-systems view
  - Hardware, software, users, economics, ….

- Security is only as strong as the weakest link
  - May have been difficult to break into the building
  - But if the data is sent unencrypted…

- Securing a system can be difficult
  - Interdisciplinary (software, hardware, UI design)
  - Humans are in the loop

- Security through obscurity does not work
  - Especially for high-value assets
  - It's only a matter of time until someone finds out

The Goals of CMSC 414

Be able to eliminate bugs and design flaws
and/or make them harder to exploit.

Be able to think like attackers.

Develop a foundation for deeply understanding
the systems we use and build.

Software **Hardware** **Protocols**

Users **Law** **Economics**

Software security

- Security is a form of dependability
  - Does the code do “what it should”
  - To this end, we follow the software lifecycle

- Distinguishing factor: an *active, malicious* attacker

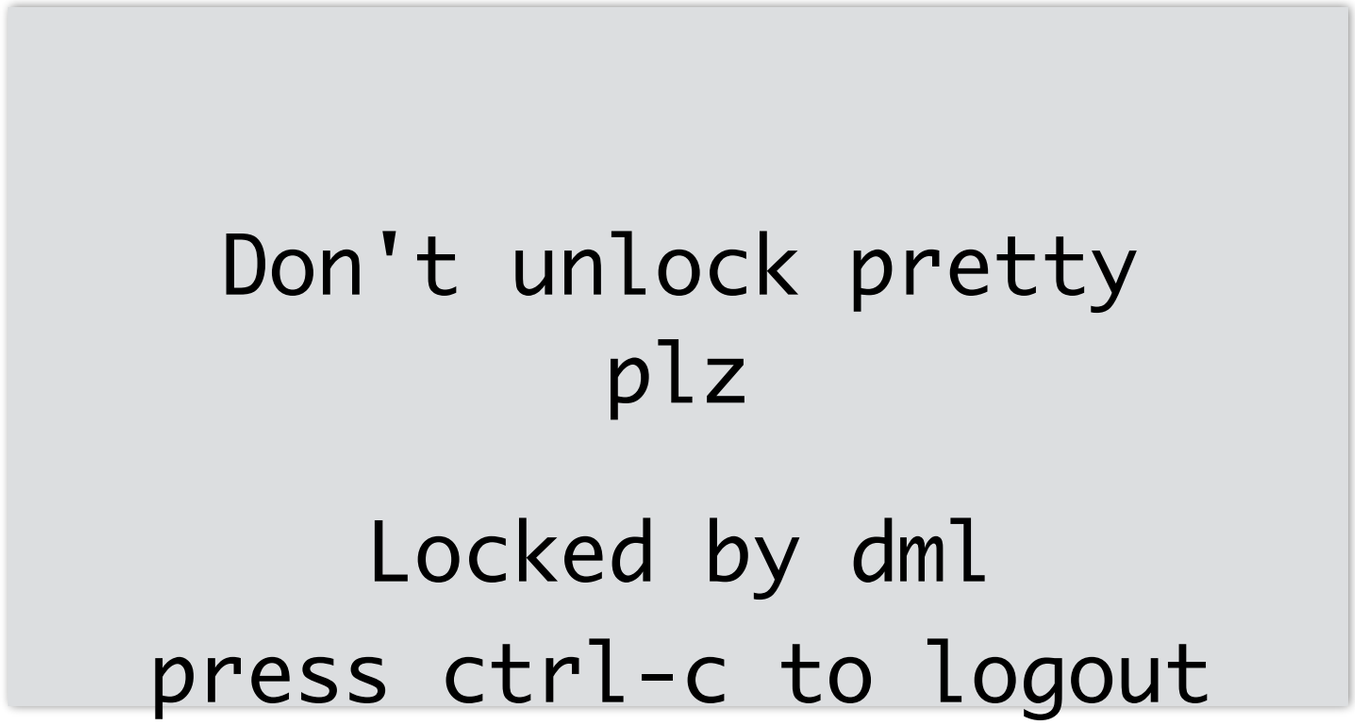
- Attack model
  - The developer is trusted
  - But the attacker can provide any inputs
    - Malformed strings
    - Malformed packets
    - etc.

What harm could an attacker possibly cause?

```
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”



```
screensaver --prompt="Don't unlock plz SP SP SP \
SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP SP"
```

Don't unlock plz

Locked by dml

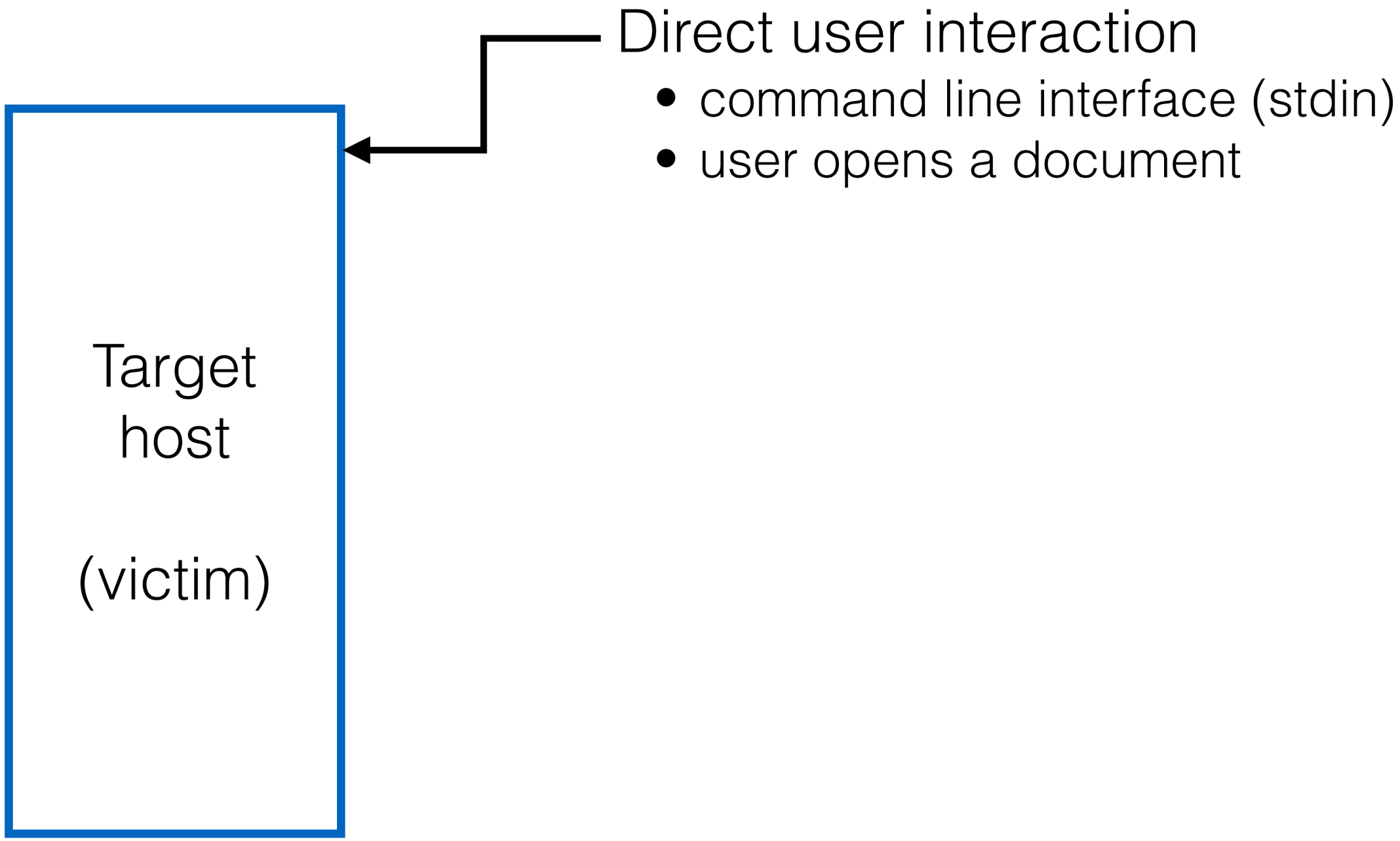
press ctrl-c to logout

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

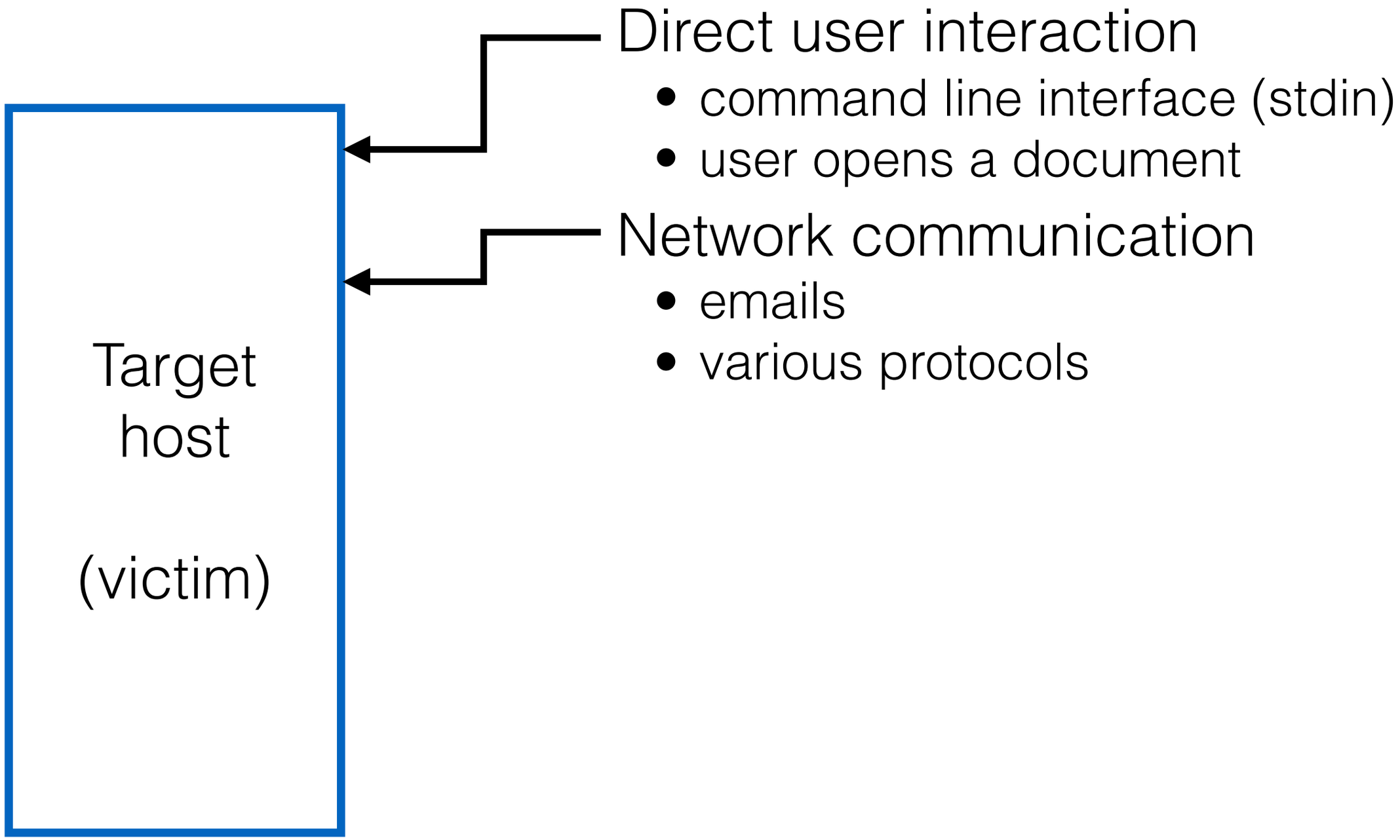
```
Under maintenance;
Do not interrupt
```

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

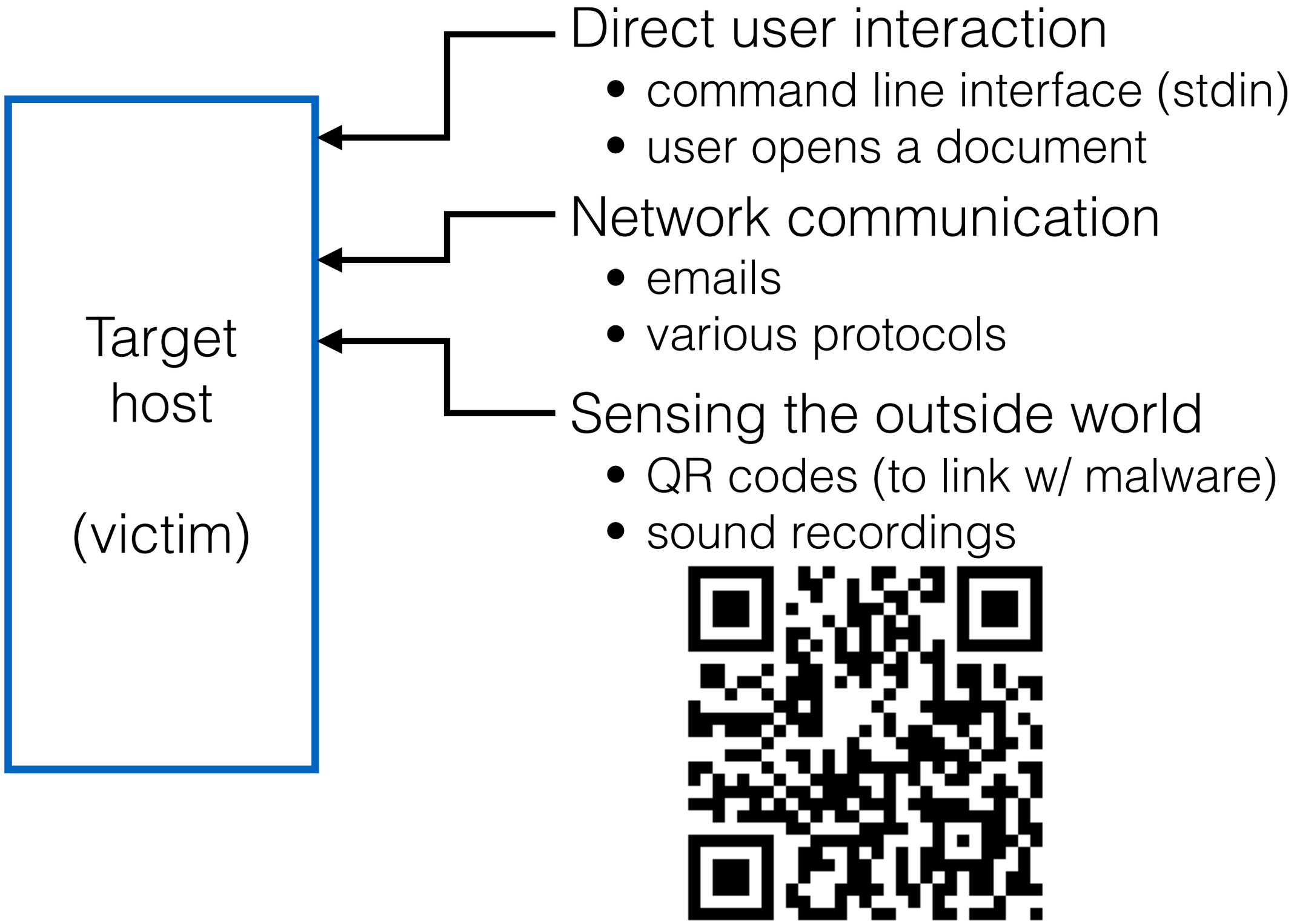
Most (interesting) software takes input



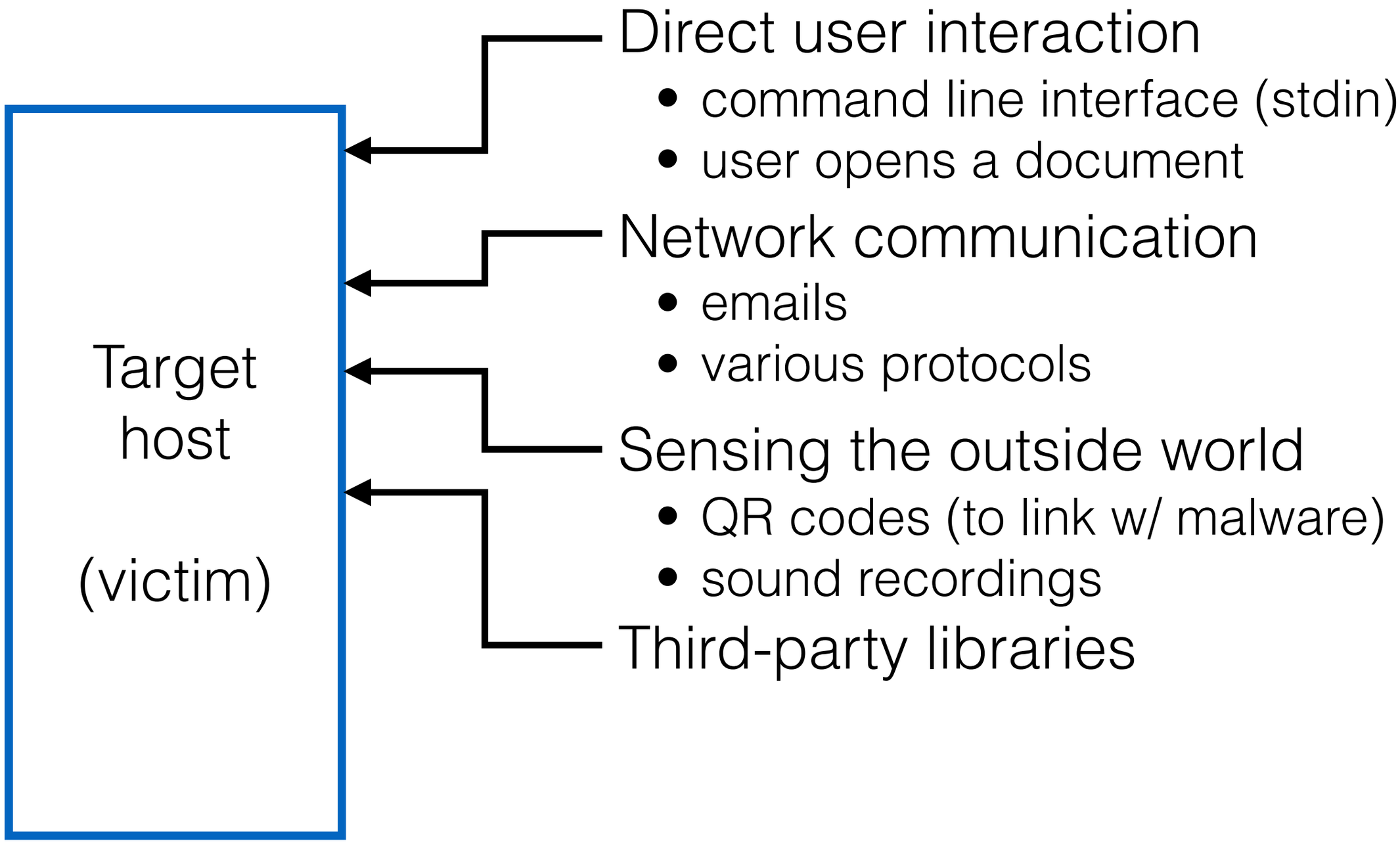
Most (interesting) software takes input



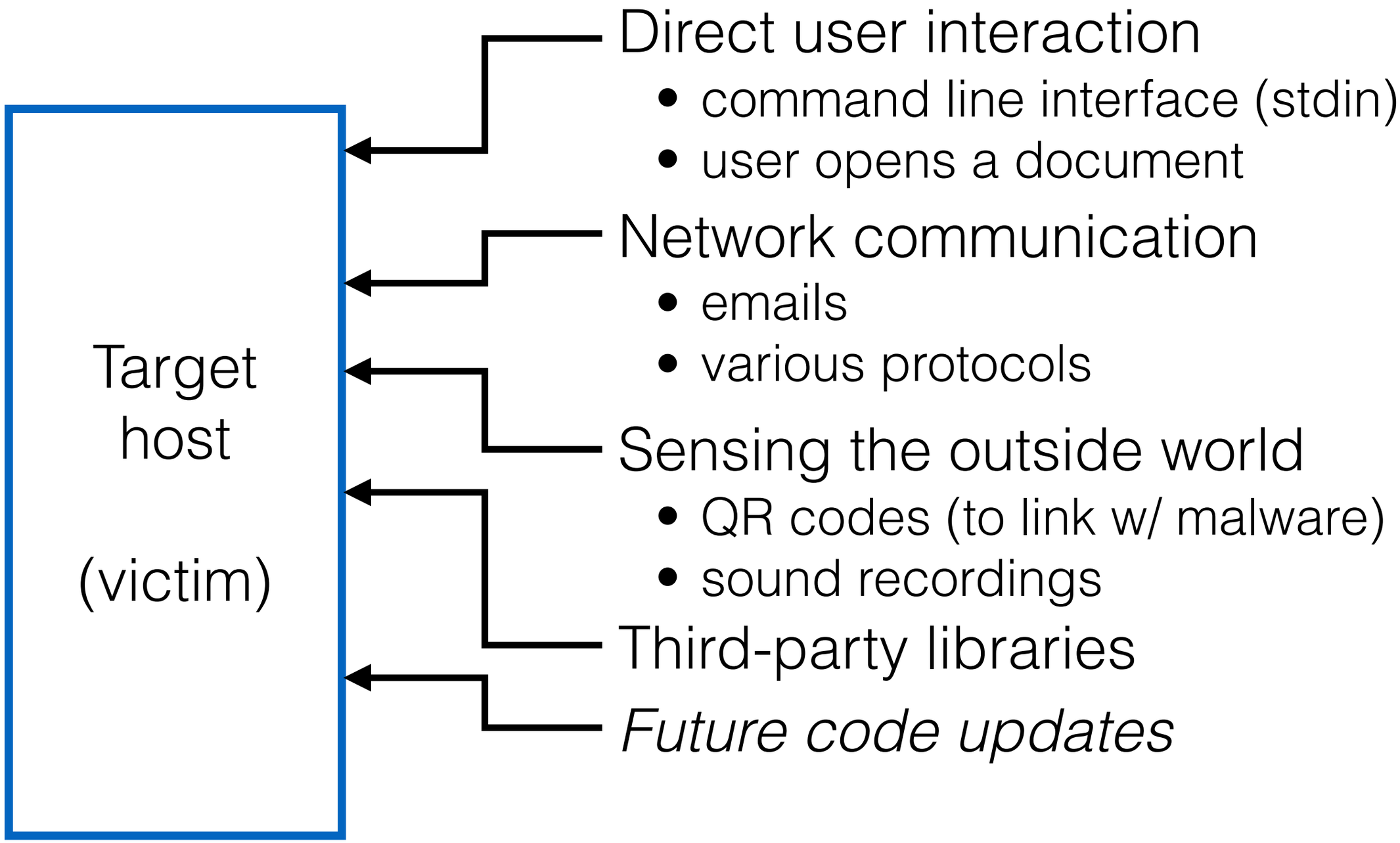
Most (interesting) software takes input



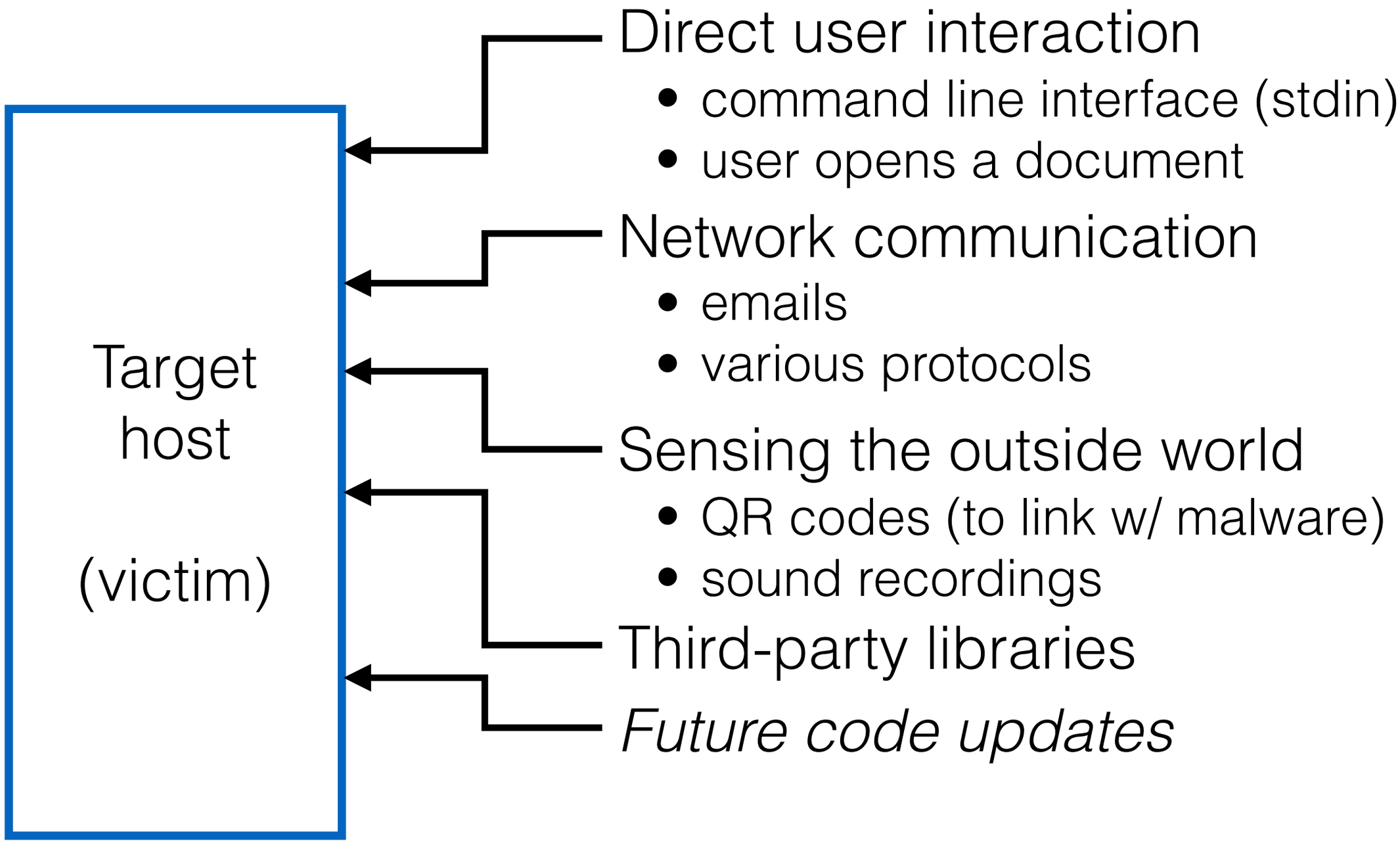
Most (interesting) software takes input



Most (interesting) software takes input

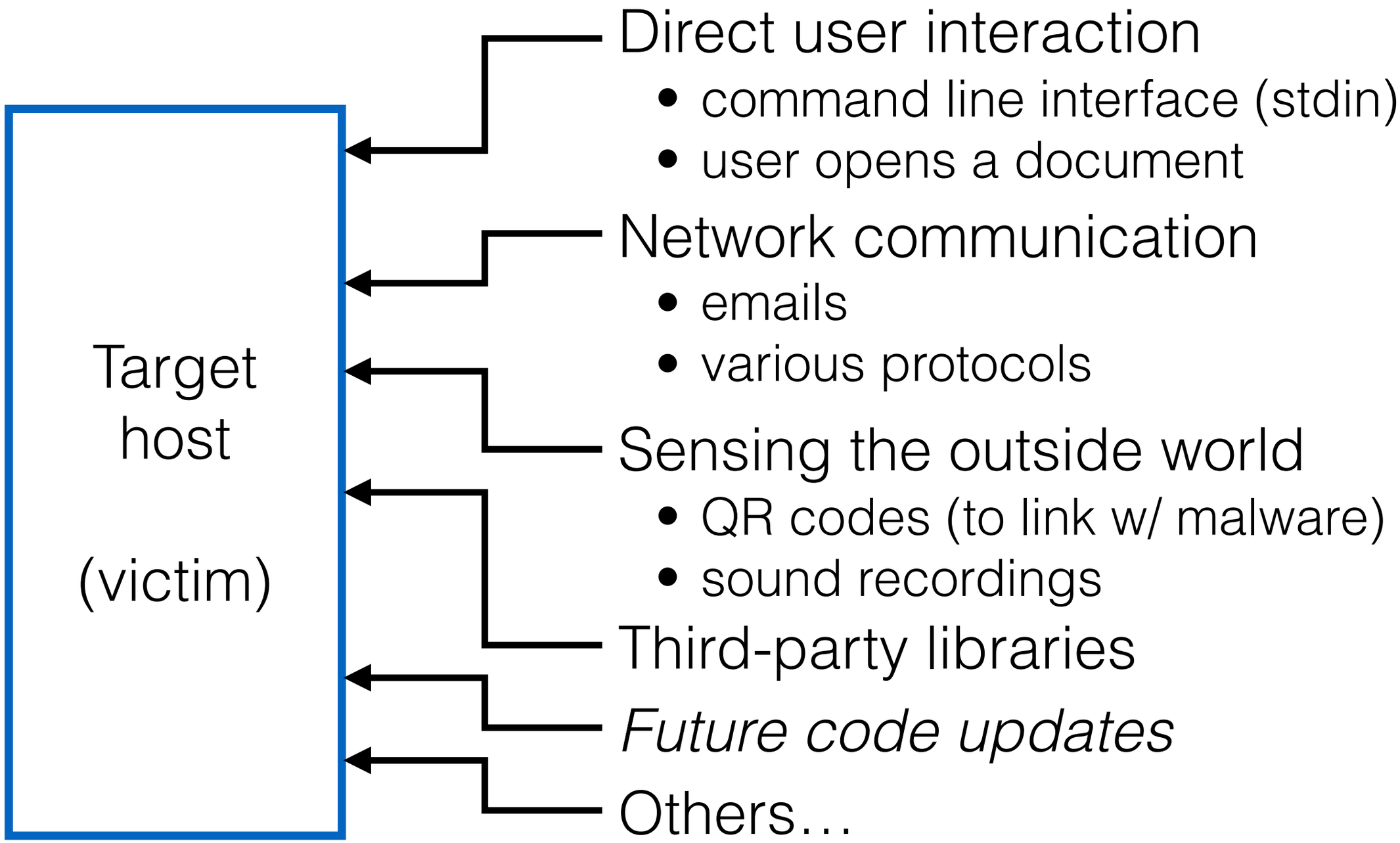


Most (interesting) software takes input



Goal: Correct operation despite malicious inputs

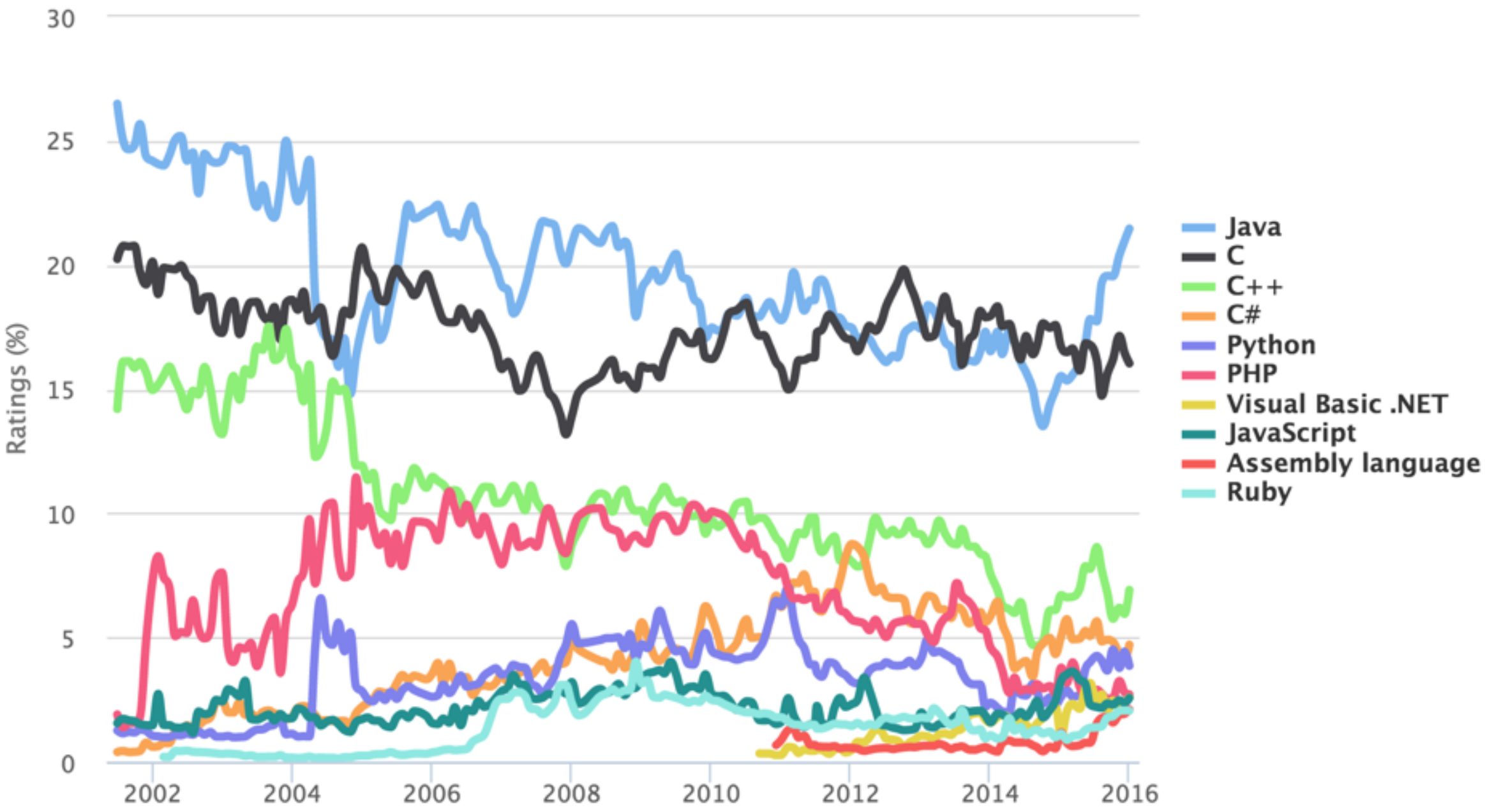
Most (interesting) software takes input



Goal: Correct operation despite malicious inputs

We’re going to focus on C

C is consistently in wide use



http://www.tiobe.com

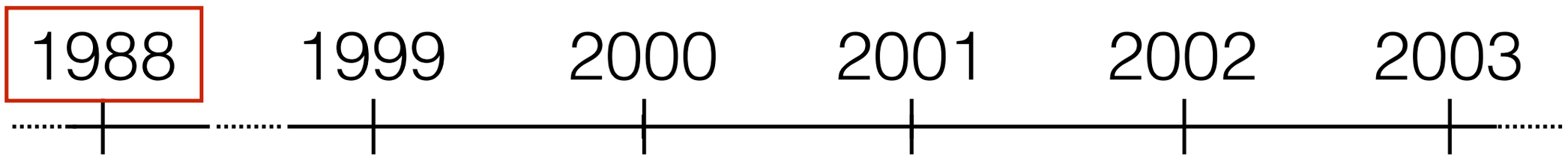
We're going to focus on C

Many mission critical systems are written in C

- Most kernels & OS utilities
  - `fingerd`
  - X windows server
- Many high-performance servers
  - Microsoft IIS
  - Microsoft SQL server
- Many embedded systems
  - Mars rover
- But the techniques apply more broadly
  - Wiibrew:"Twilight Hack" exploits buffer overflow when saving the name of Link's horse, Epona

We're going to focus on C

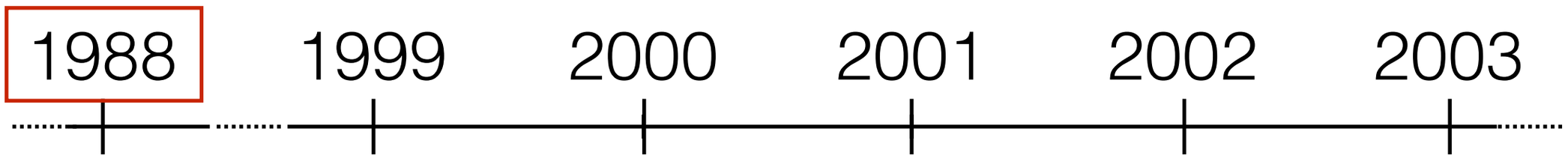
A breeding ground for *buffer overflow attacks*



- 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

We're going to focus on C

A breeding ground for *buffer overflow attacks*

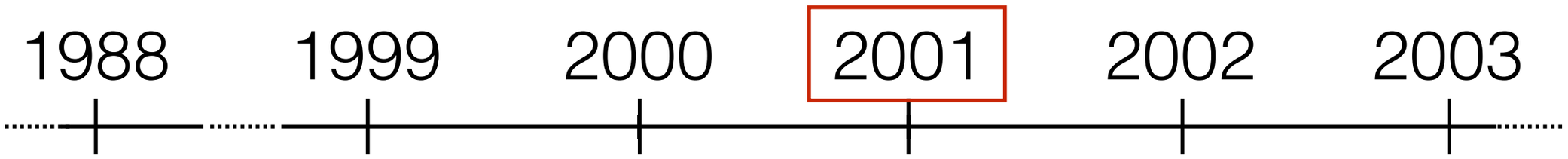


- Morris worm
  - Propagated across machines (too aggressively, thanks to a bug)
  - One way it propagated was a **buffer overflow attack** against a vulnerable version of `fingerd` on VAXes
    - Sent a special string to the finger daemon, which caused it to execute code that created a new worm copy
    - 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

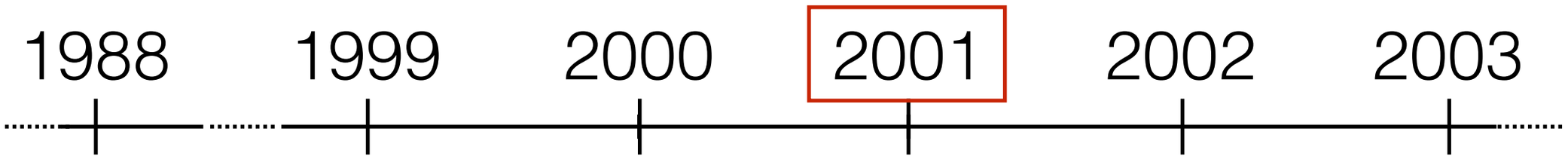
A breeding ground for *buffer overflow attacks*



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

We're going to focus on C

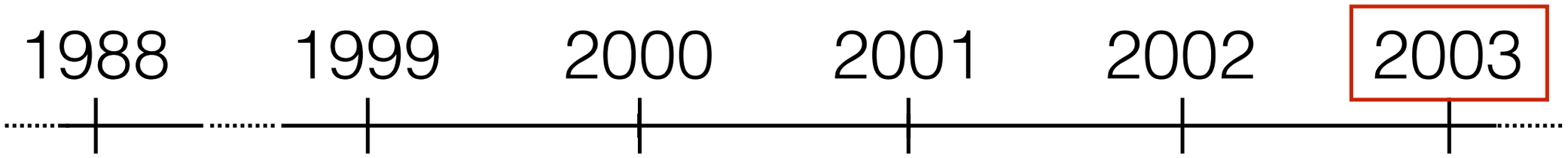
A breeding ground for *buffer overflow attacks*



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

We're going to focus on C

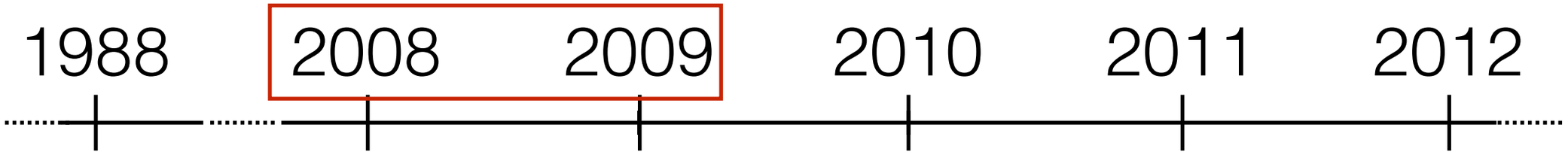
A breeding ground for *buffer overflow attacks*



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

We’re going to focus on C

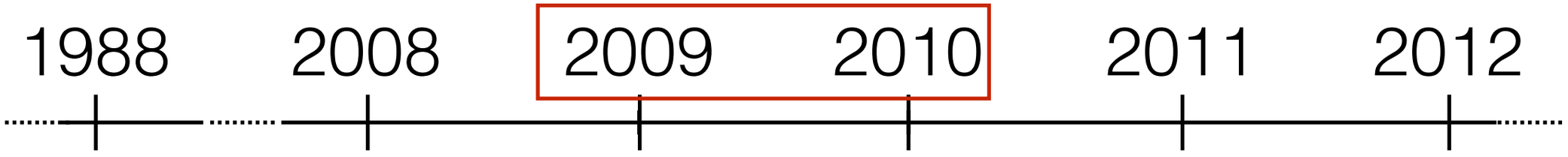
A breeding ground for *buffer overflow attacks*



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

We're going to focus on C

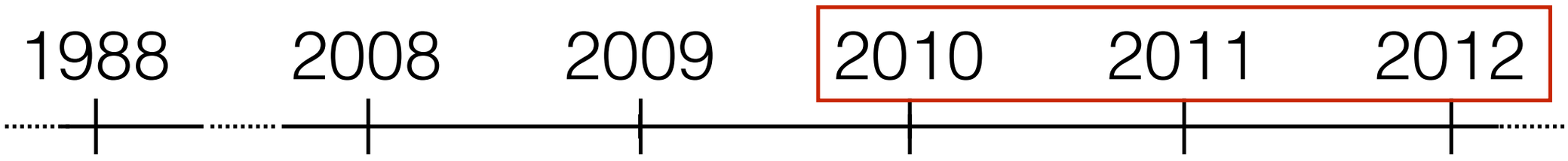
A breeding ground for *buffer overflow attacks*



- 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

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.

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

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));
```

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

Array of **int**'s

```
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

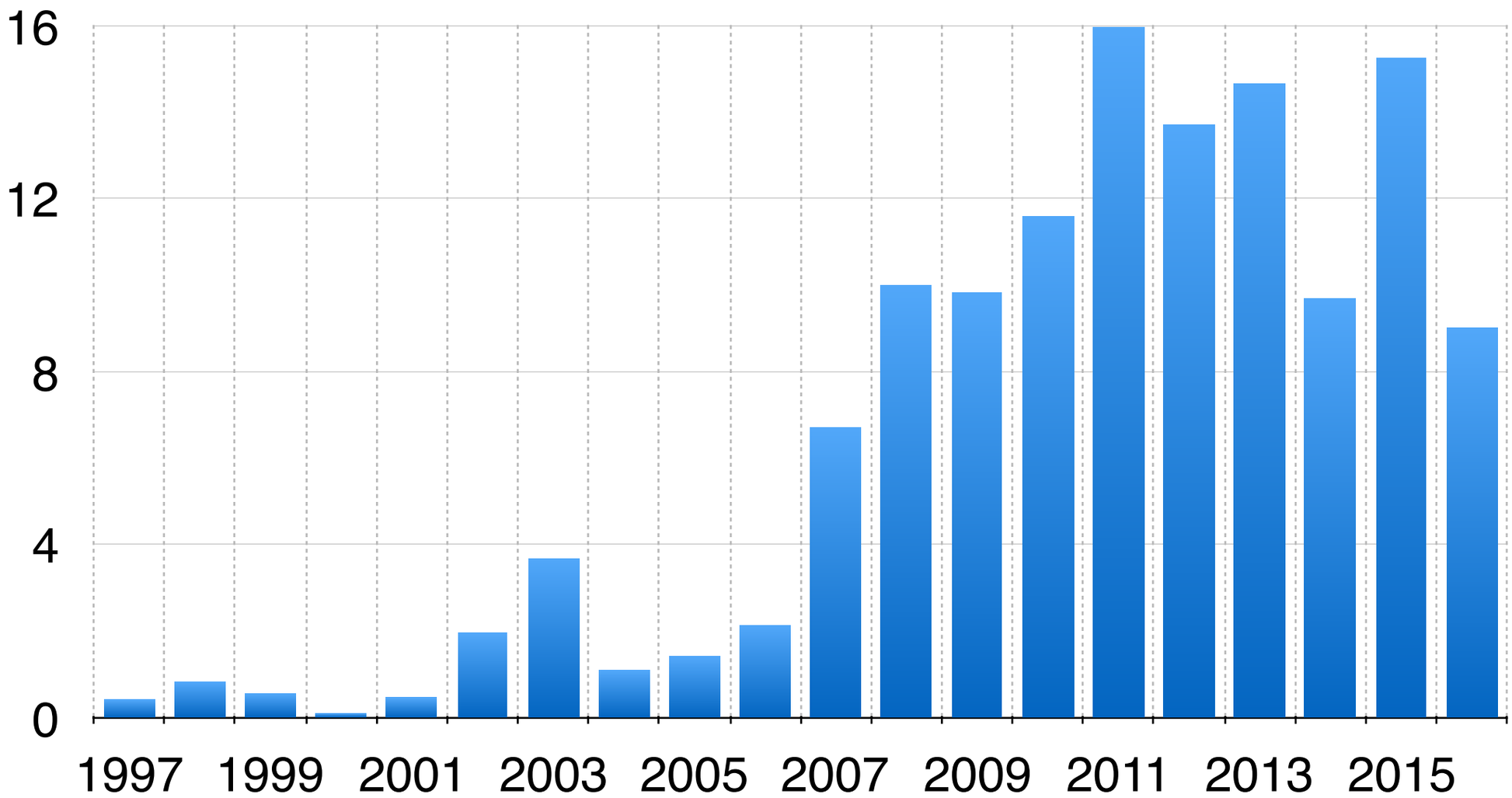
Buffer too small

Writes beyond the buffer

```
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 overflows are prevalent

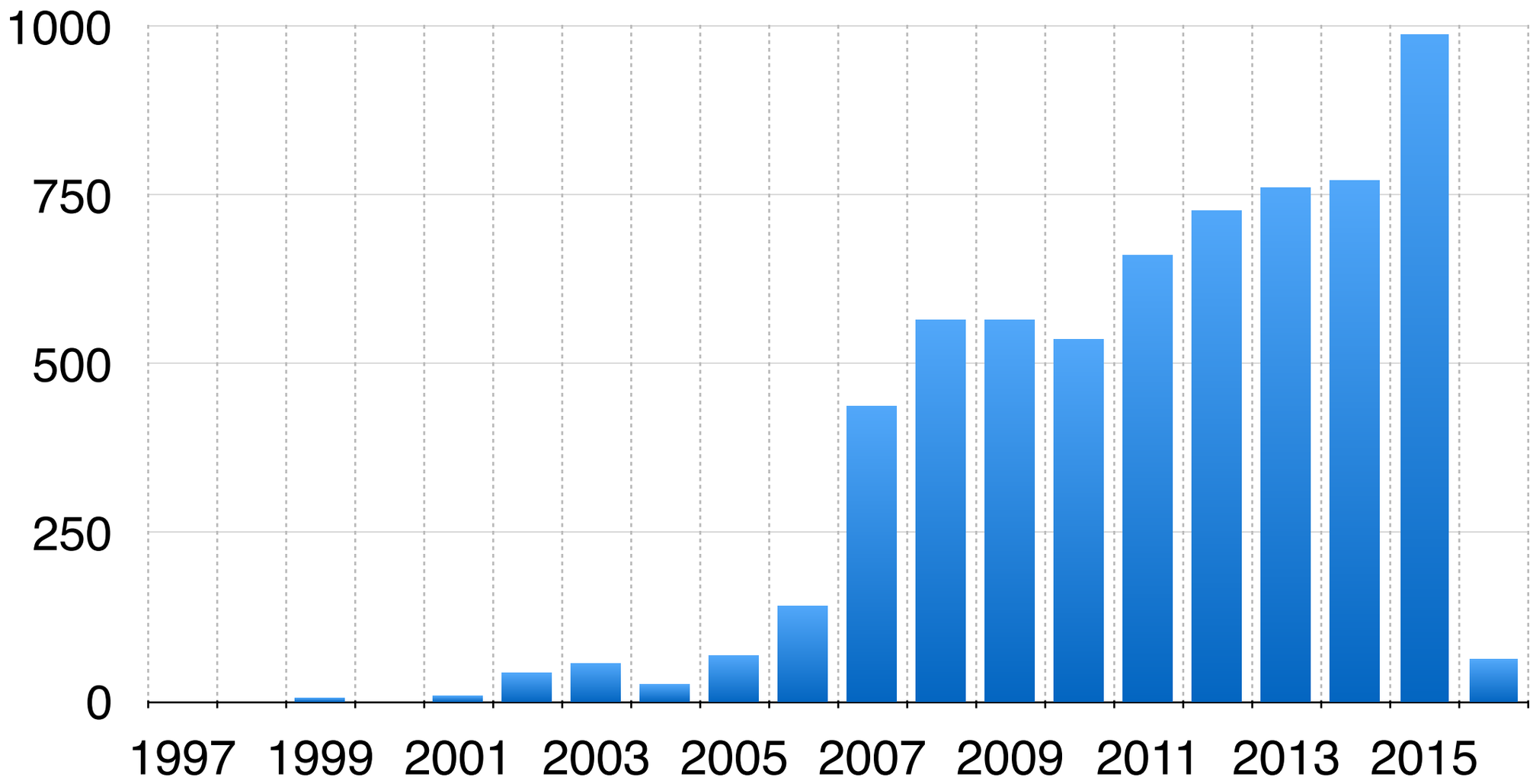
Significant percent of *all* vulnerabilities



Data from the National Vulnerability Database

Buffer overflows are prevalent

Total number of buffer overflow vulnerabilities



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)

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

E-voting

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

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)

Later

This class

Later

E-voting

Later

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

Our goals

- Understand how these attacks work, and how to defend against them

- These require knowledge about:
  - The compiler
  - The OS
  - The architecture

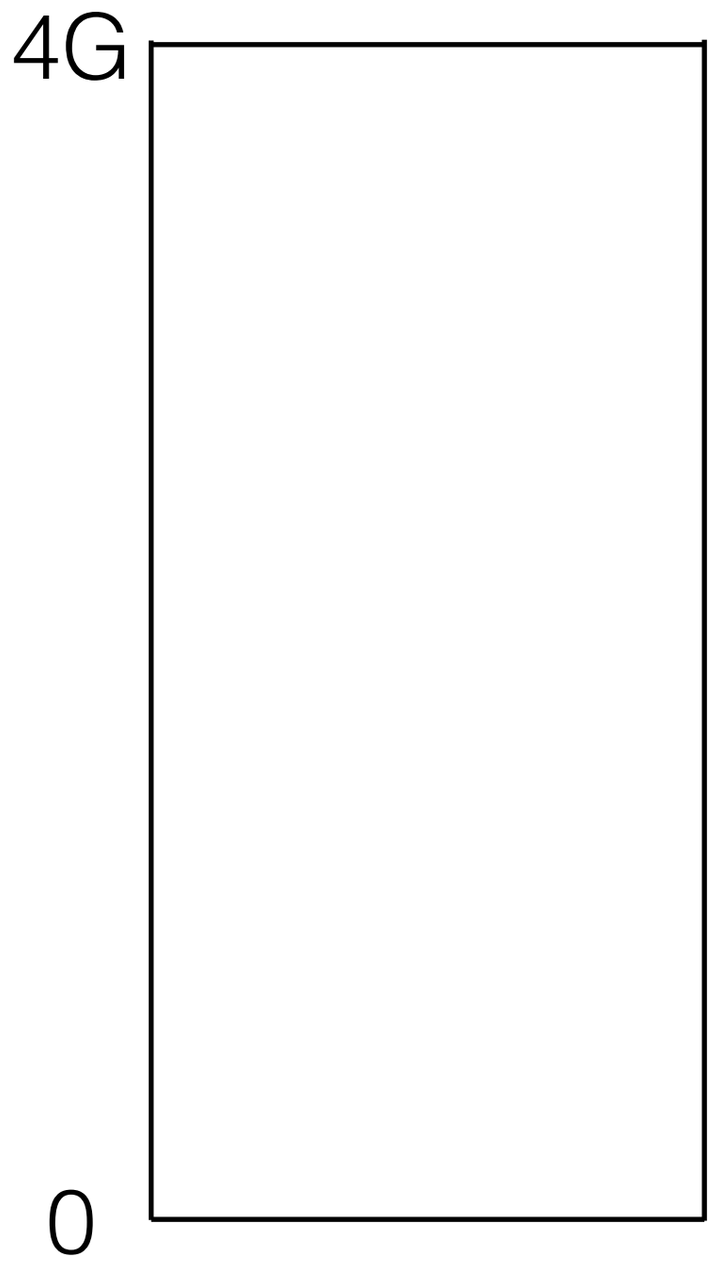
Analyzing security requires a whole-systems view

Memory layout

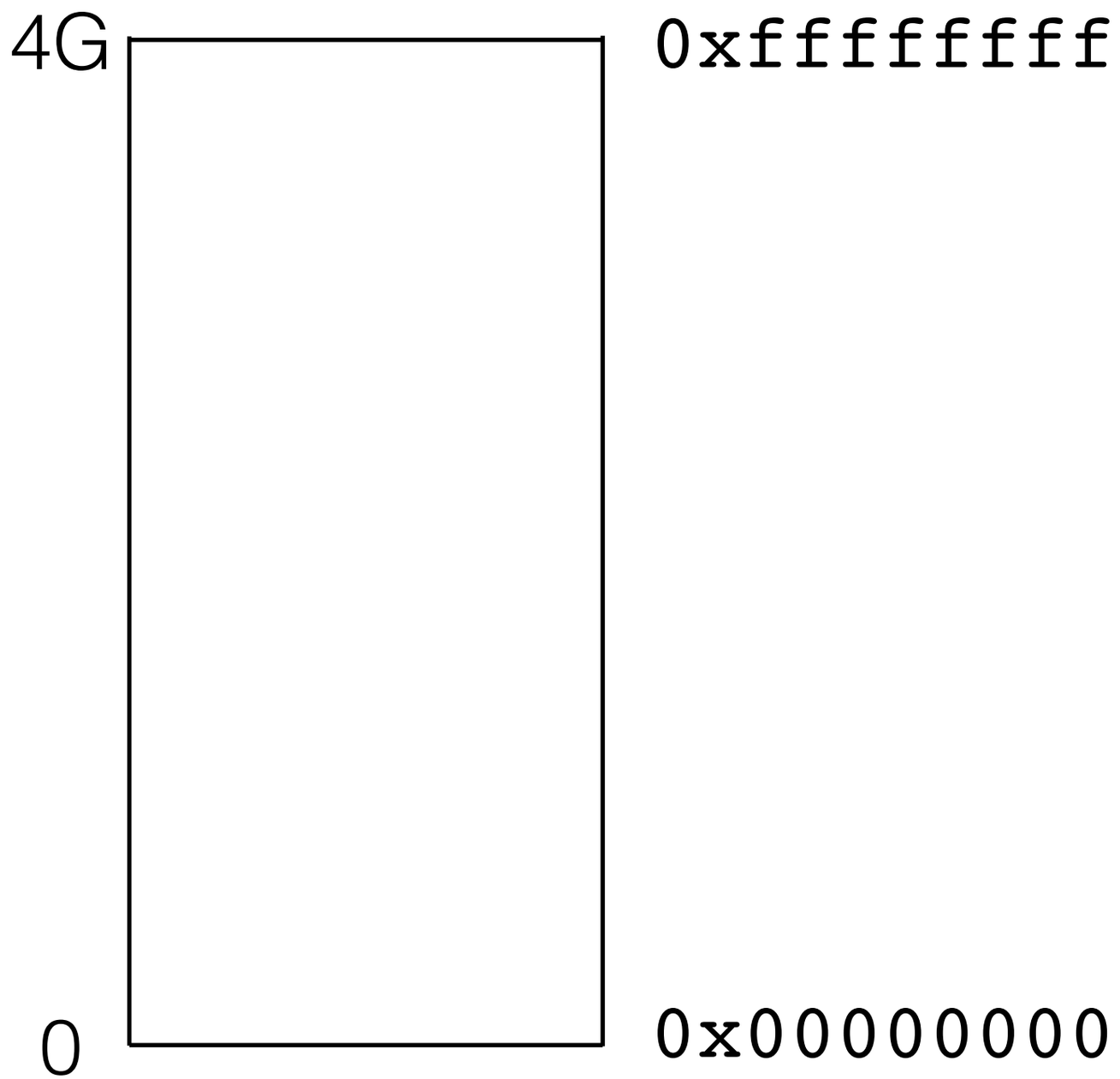
Refresher

- How is program data laid out in memory?
- What does the stack look like?
- What effect does calling (and returning from) a function have on memory?
- We are focusing on the Linux process model
  - Similar to other operating systems

All programs are stored in memory

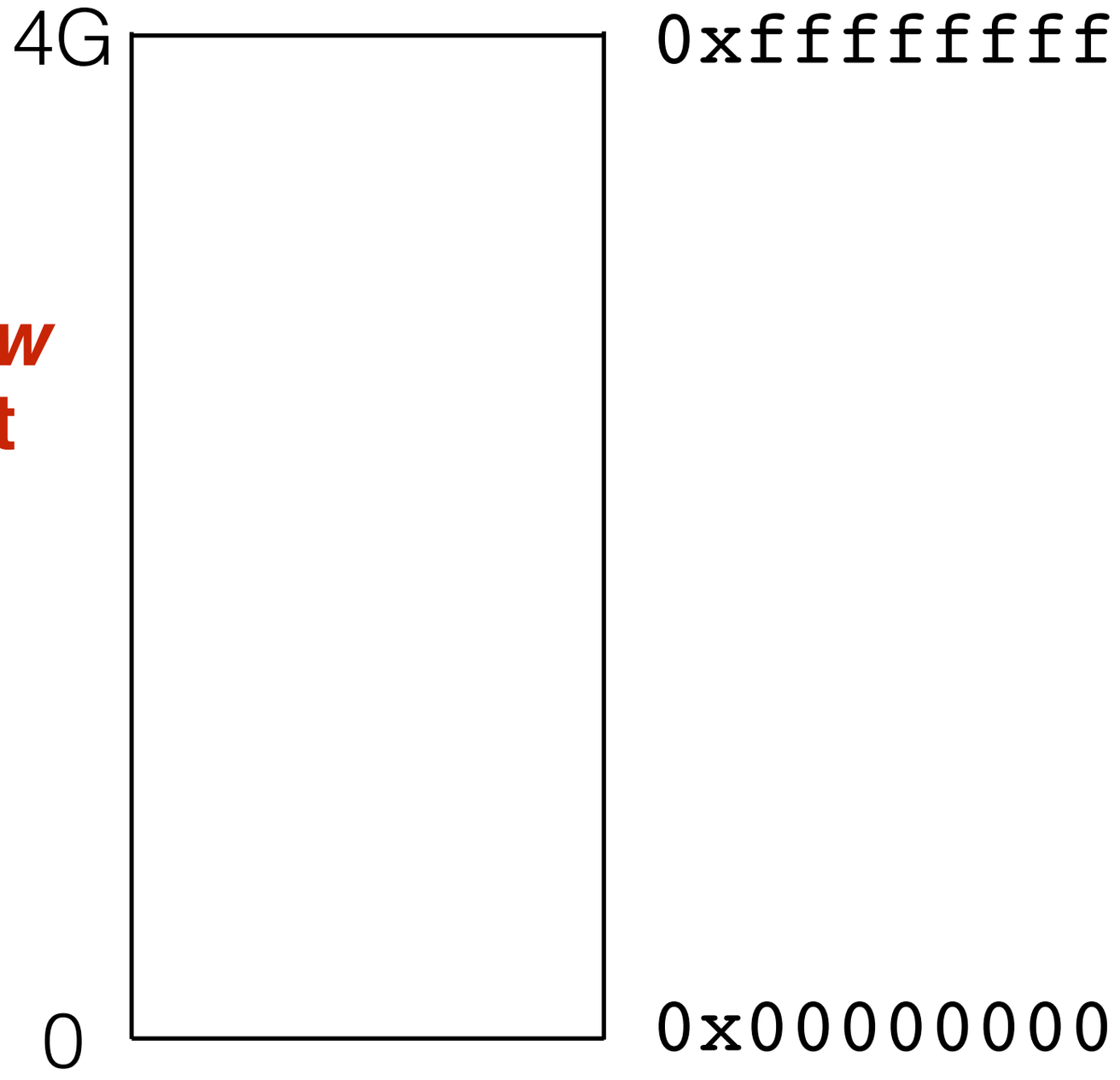


All programs are stored in memory

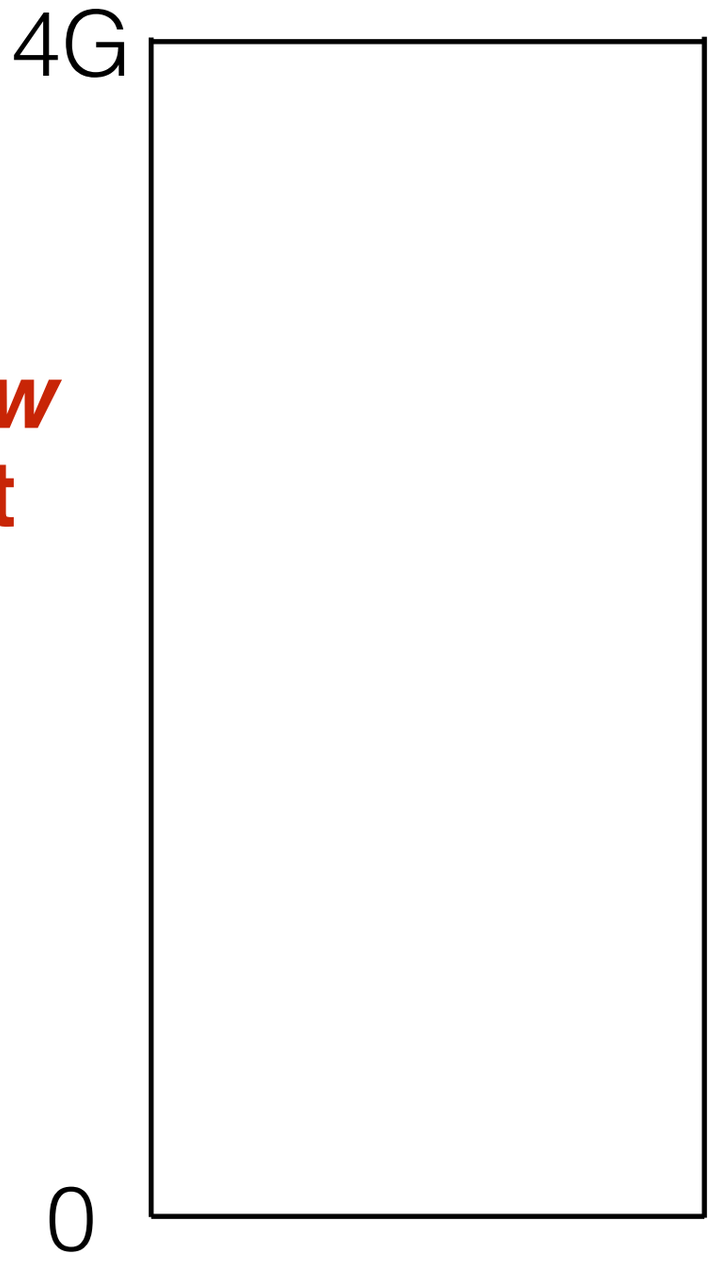


All programs are stored in memory

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



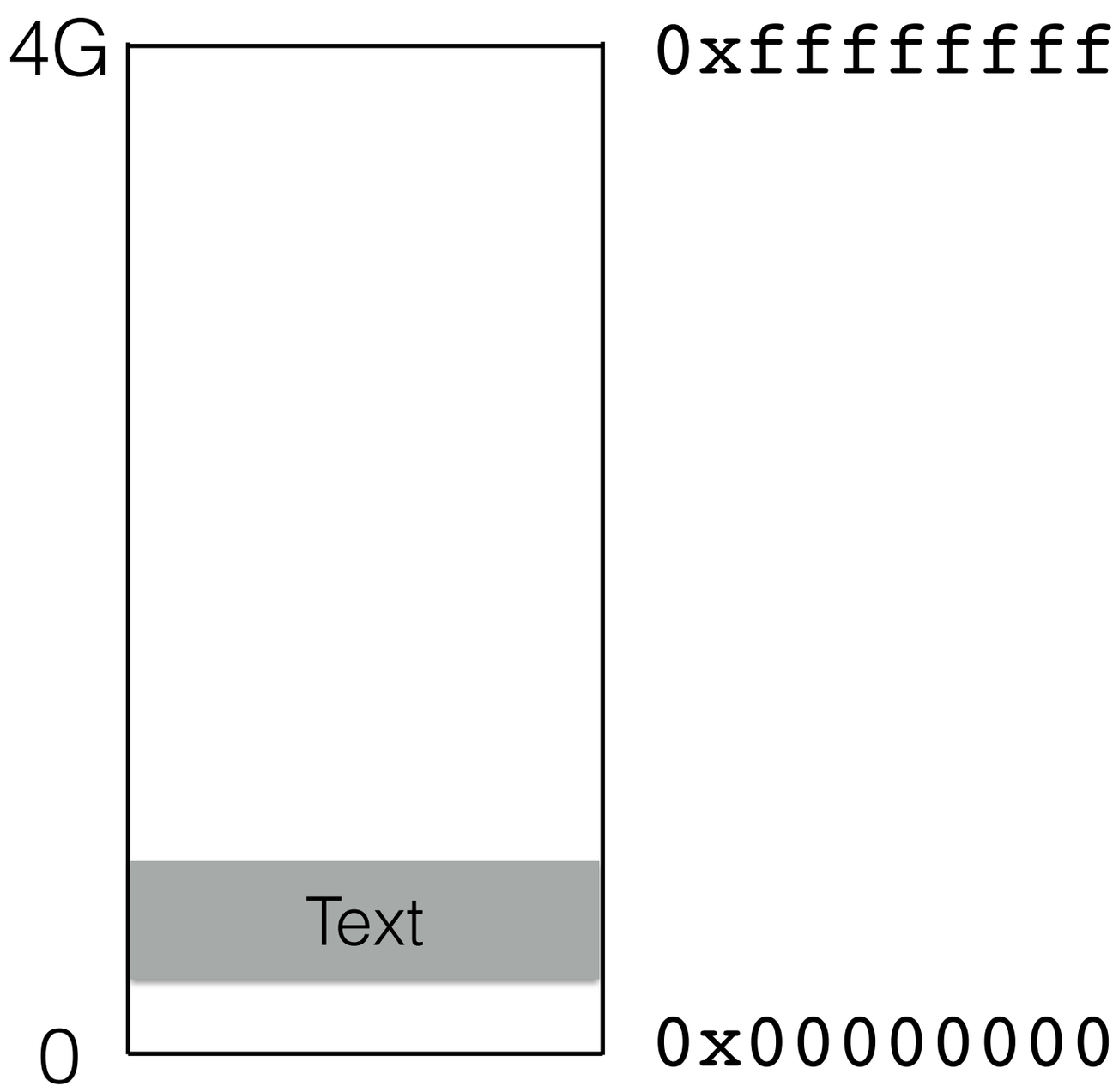
All programs are stored in memory



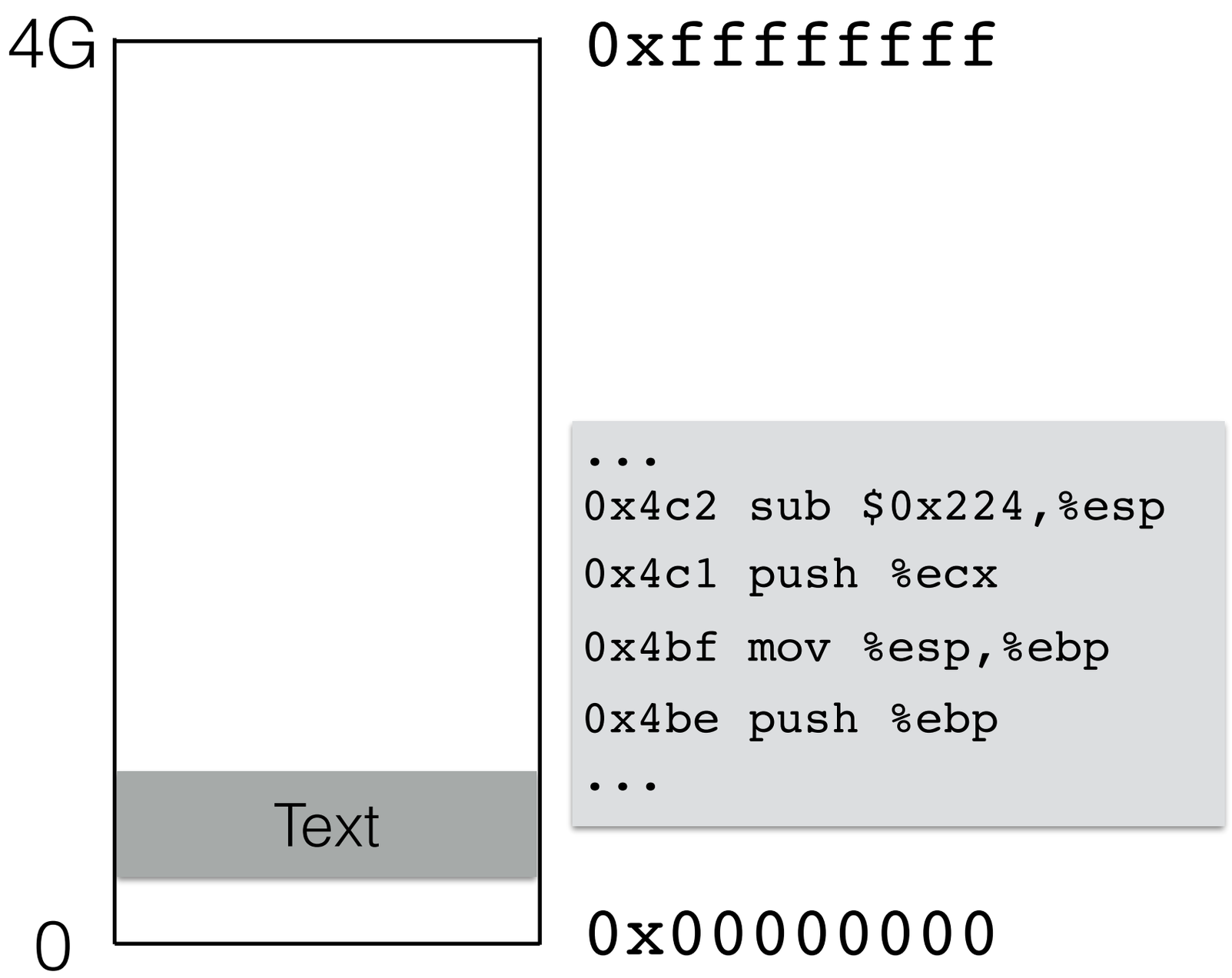
The *process's view* of memory is that it owns all of it

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

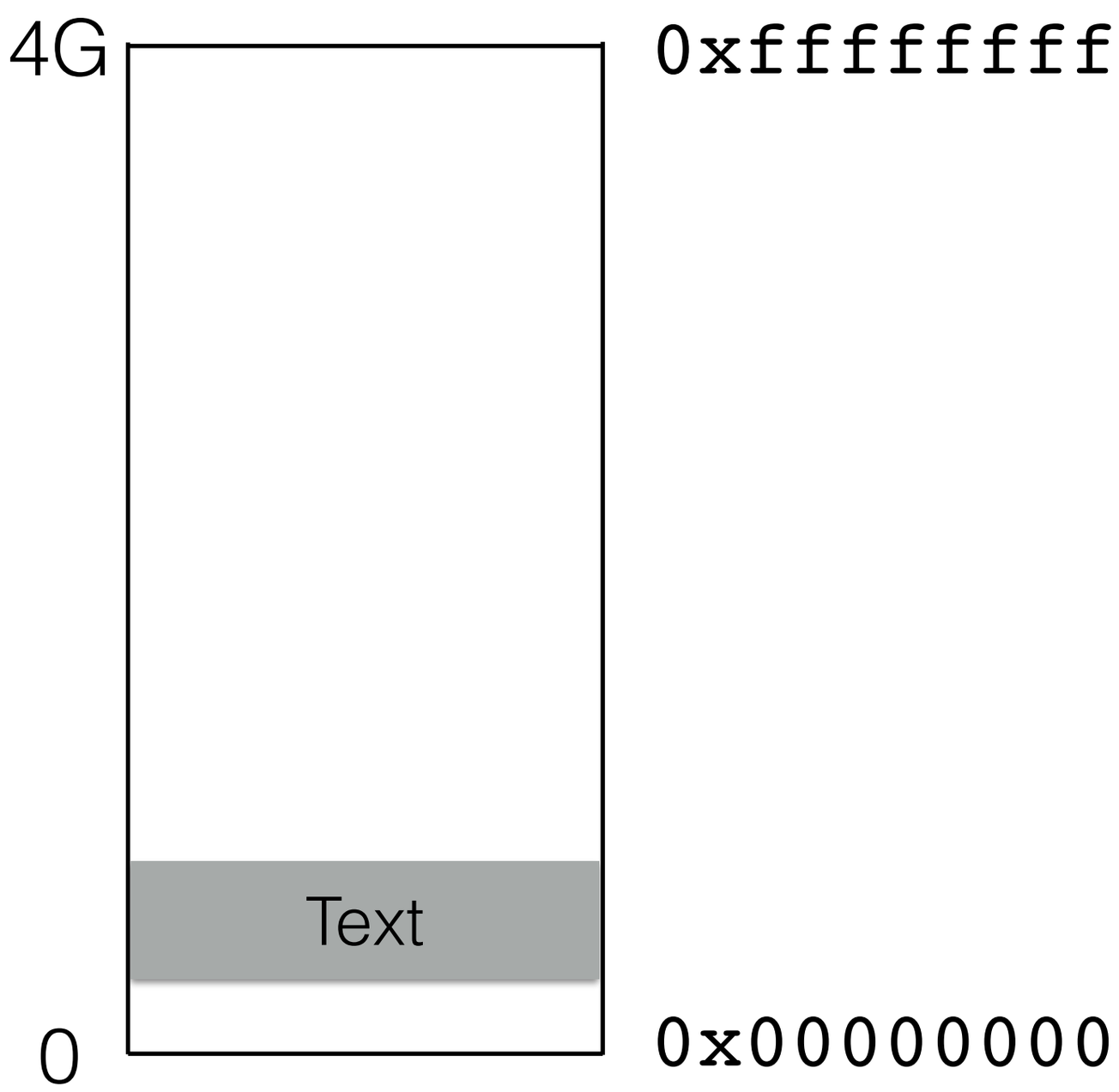
The instructions themselves are in memory



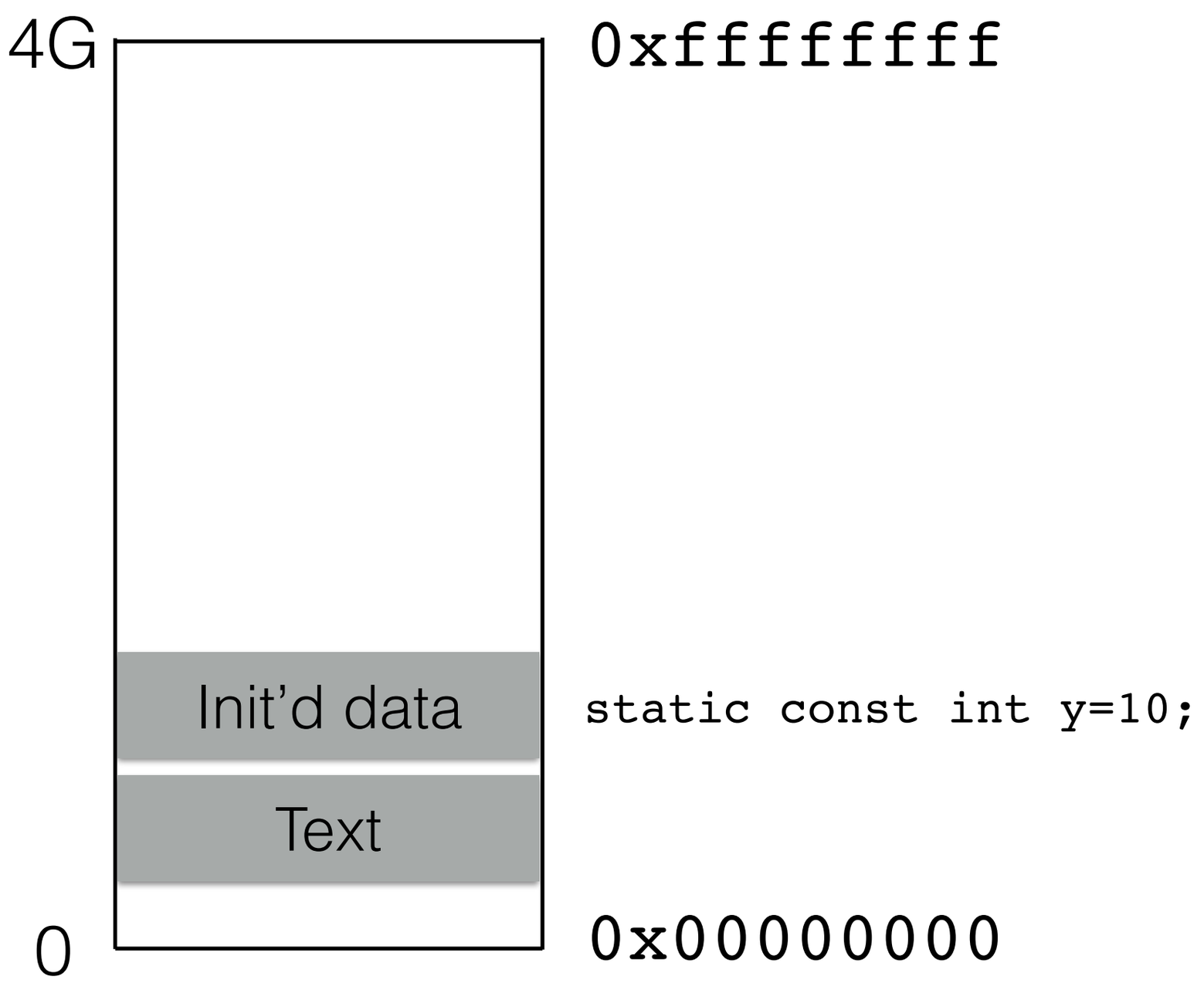
The instructions themselves are in memory



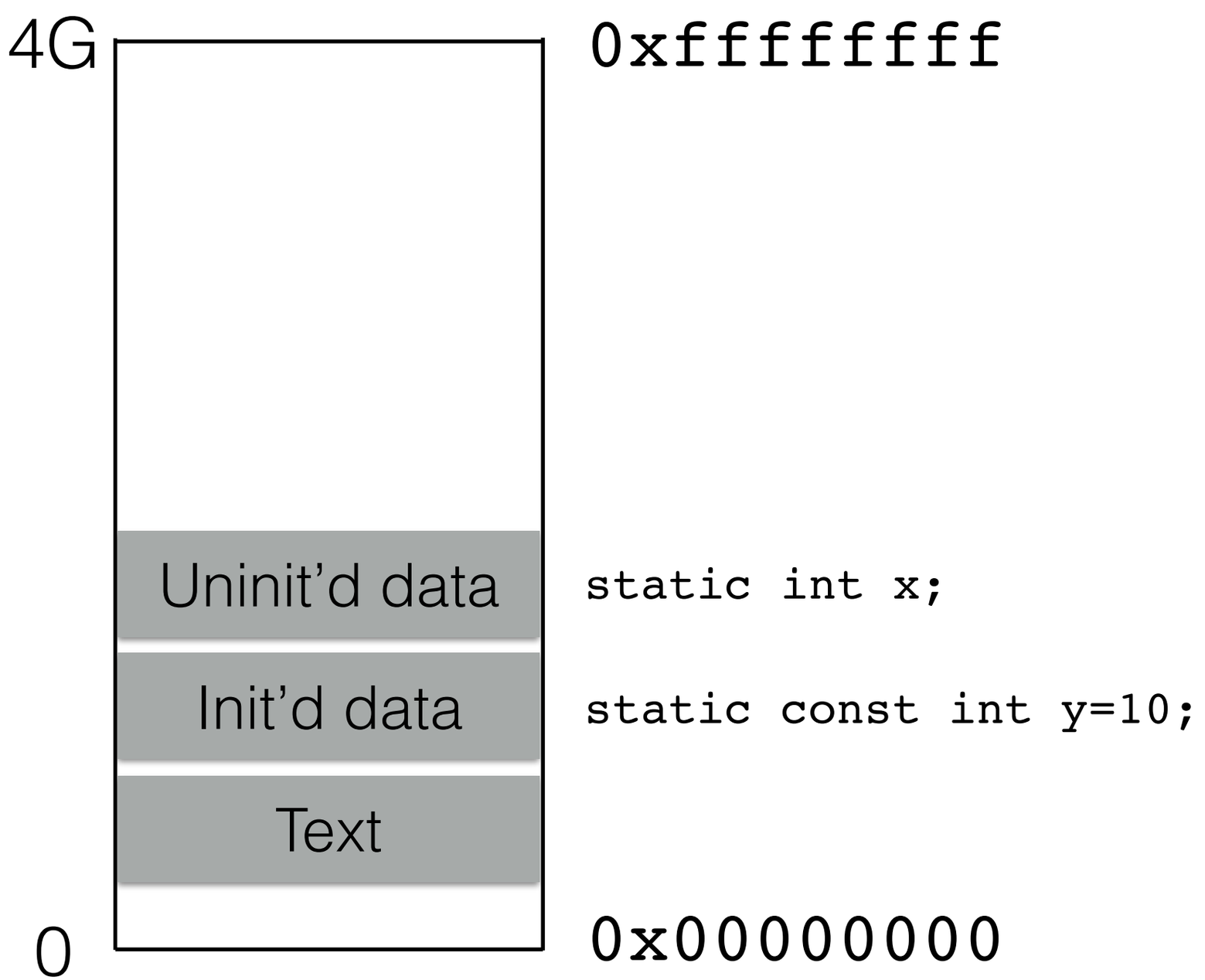
Data's location depends on how it's created



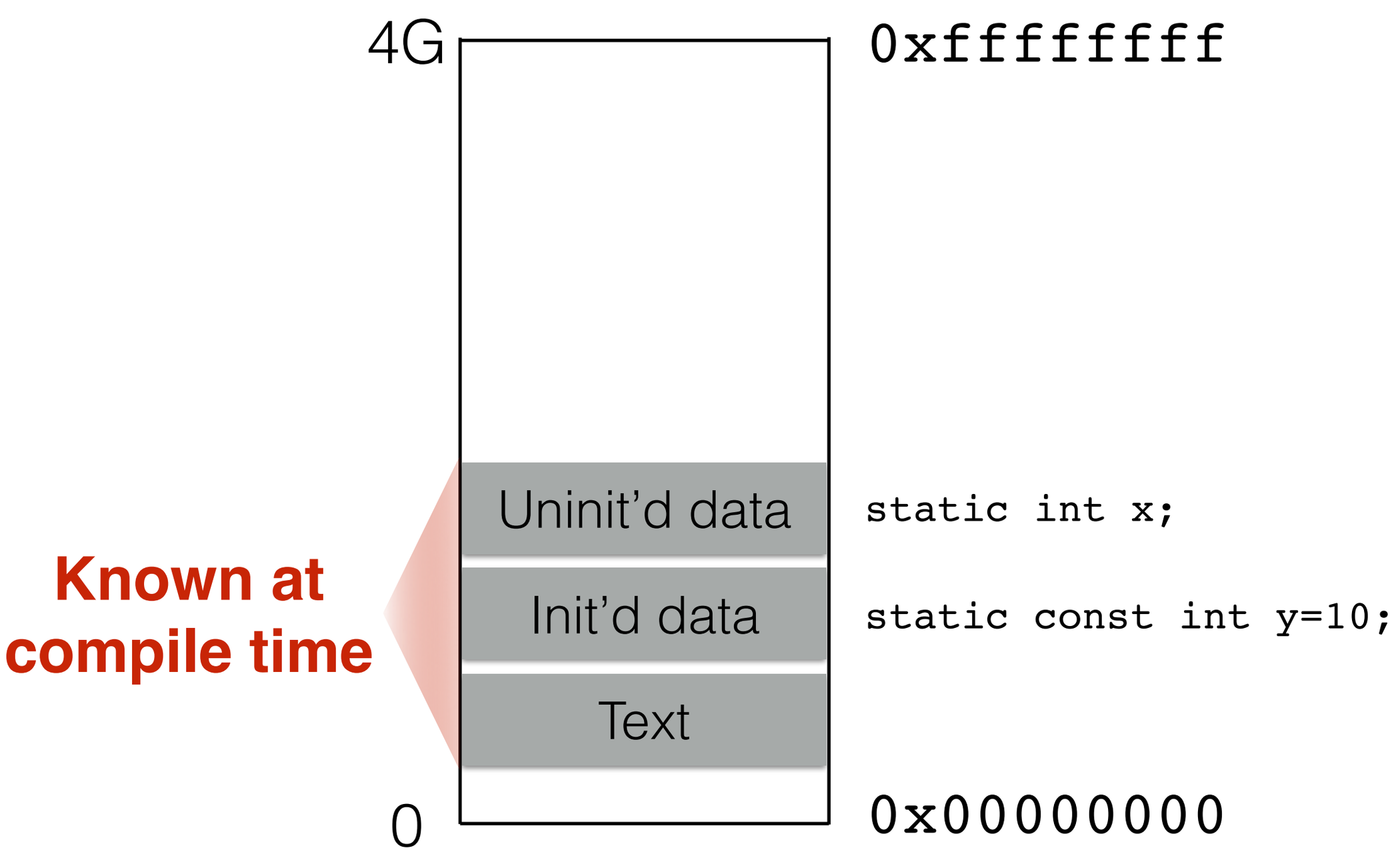
Data's location depends on how it's created



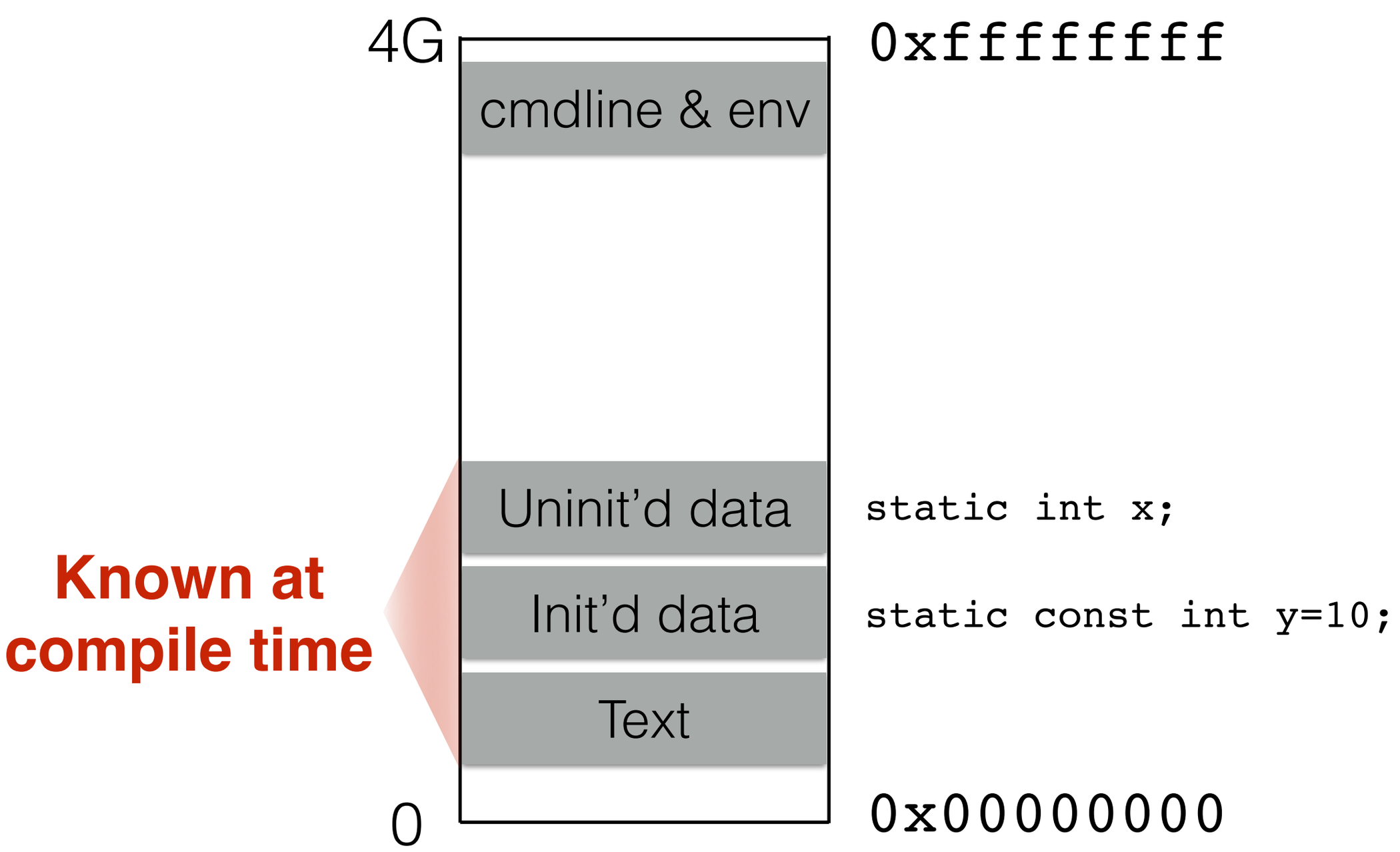
Data's location depends on how it's created



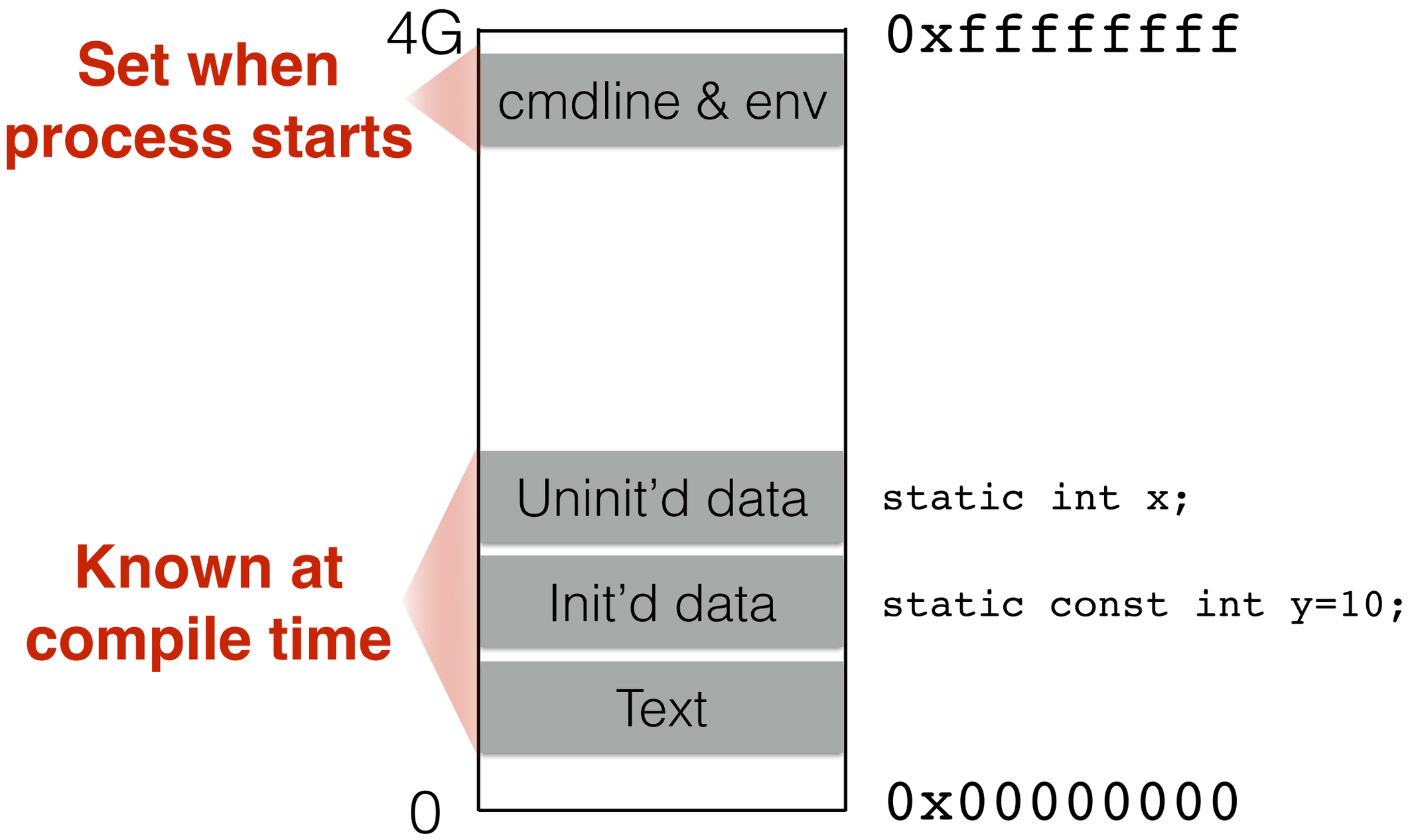
Data's location depends on how it's created



Data's location depends on how it's created



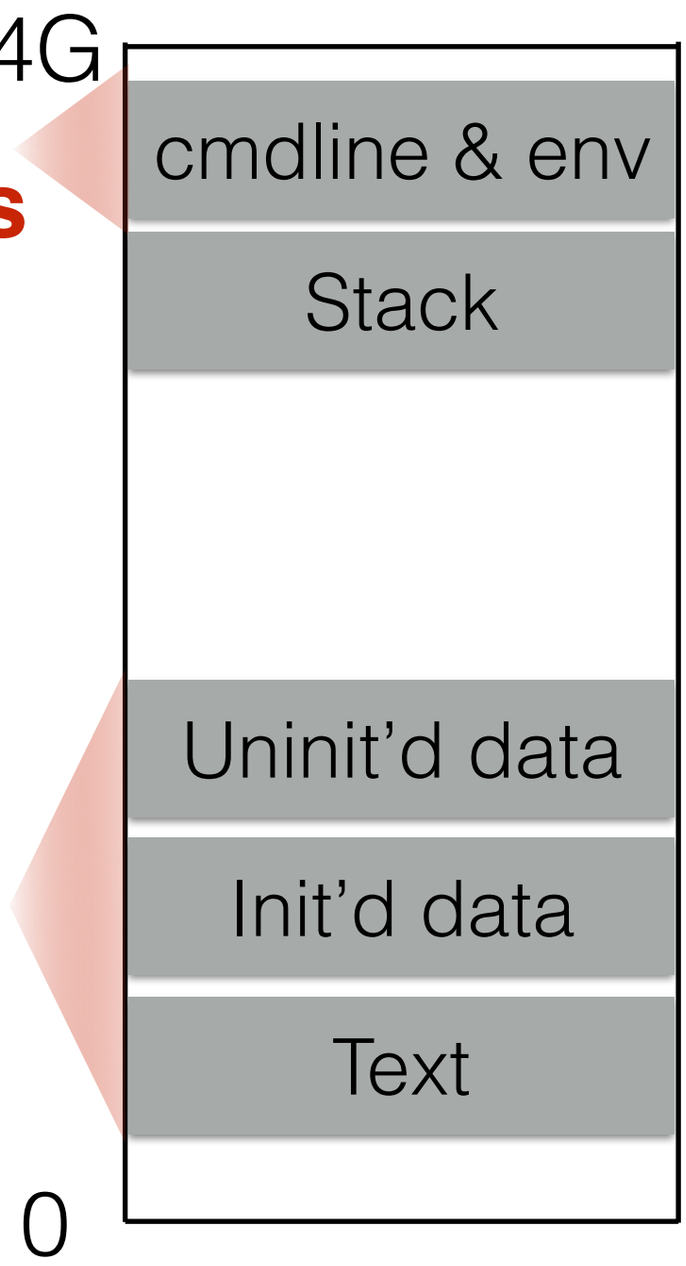
Data's location depends on how it's created



Data's location depends on how it's created

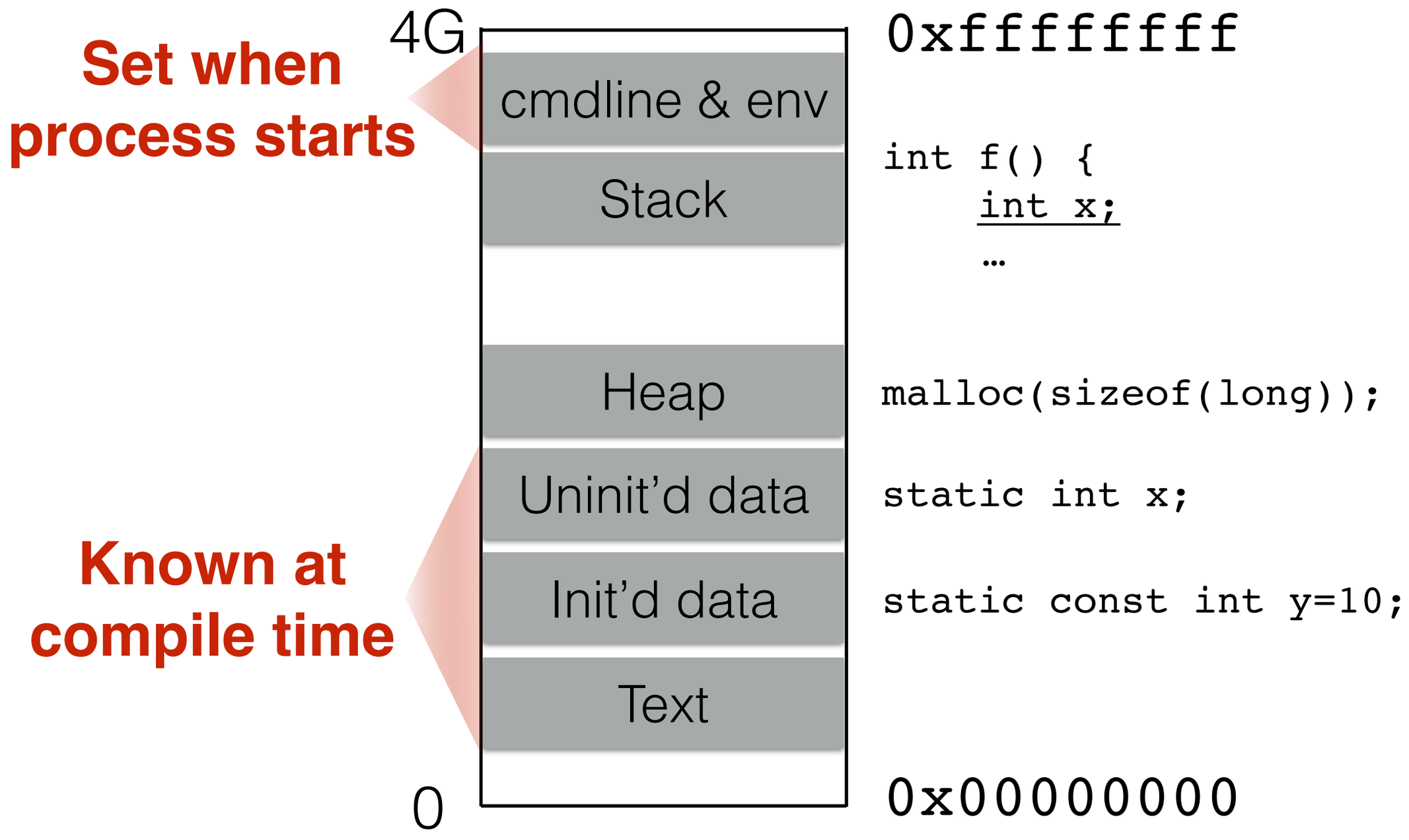
Set when process starts

Known at compile time




```
0xffffffff

int f() {
    int x;
    …

static int x;

static const int y=10;

0x00000000
```

Data's location depends on how it's created

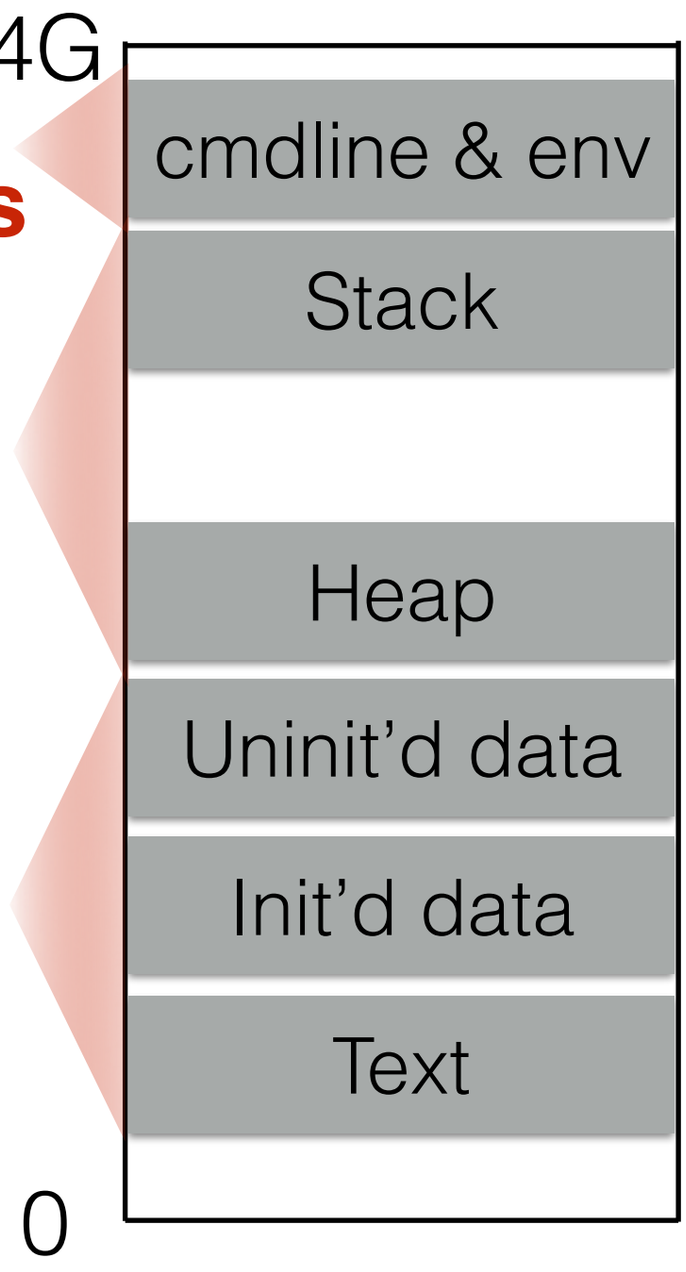


Data's location depends on how it's created

Set when process starts

Runtime

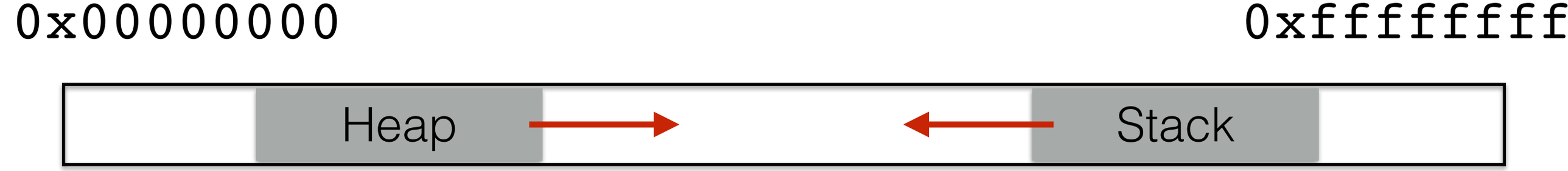
Known at compile time




```
0xffffffff

int f() {
    int x;
    …

malloc(sizeof(long));

static int x;

static const int y=10;

0x00000000
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions



We are going to focus on runtime attacks

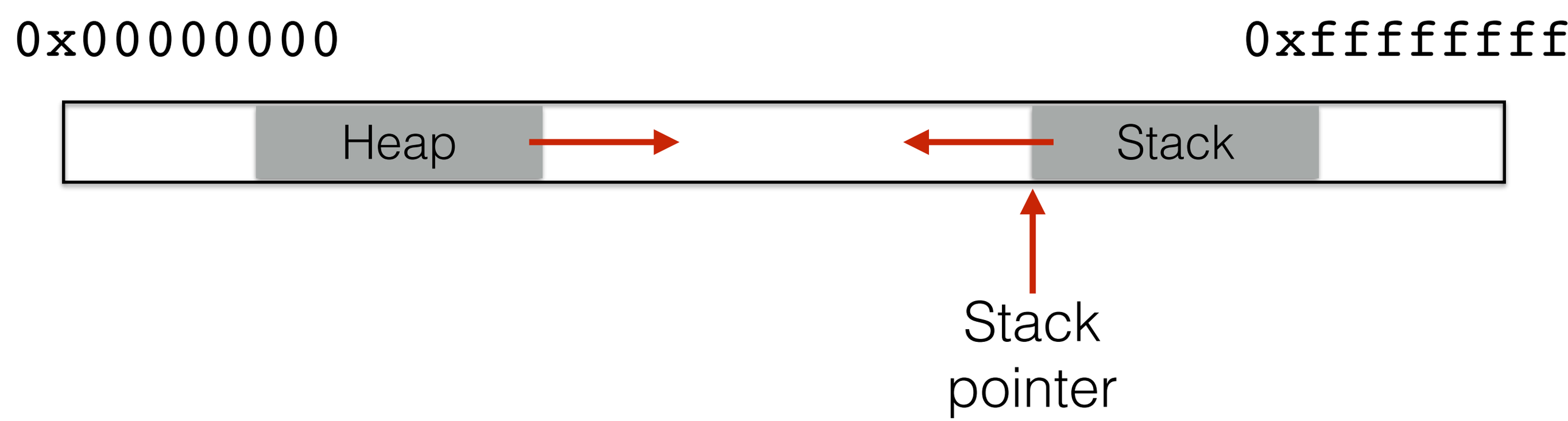
Stack and heap grow in opposite directions

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

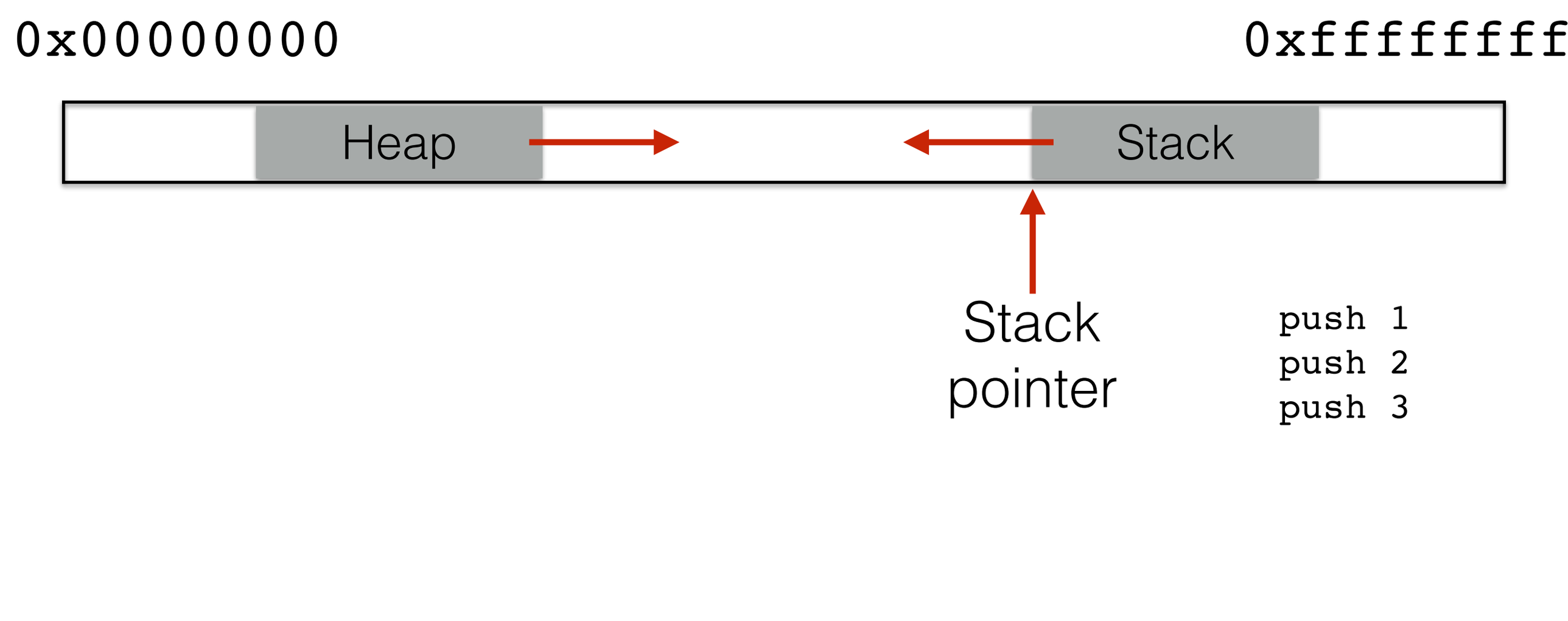
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

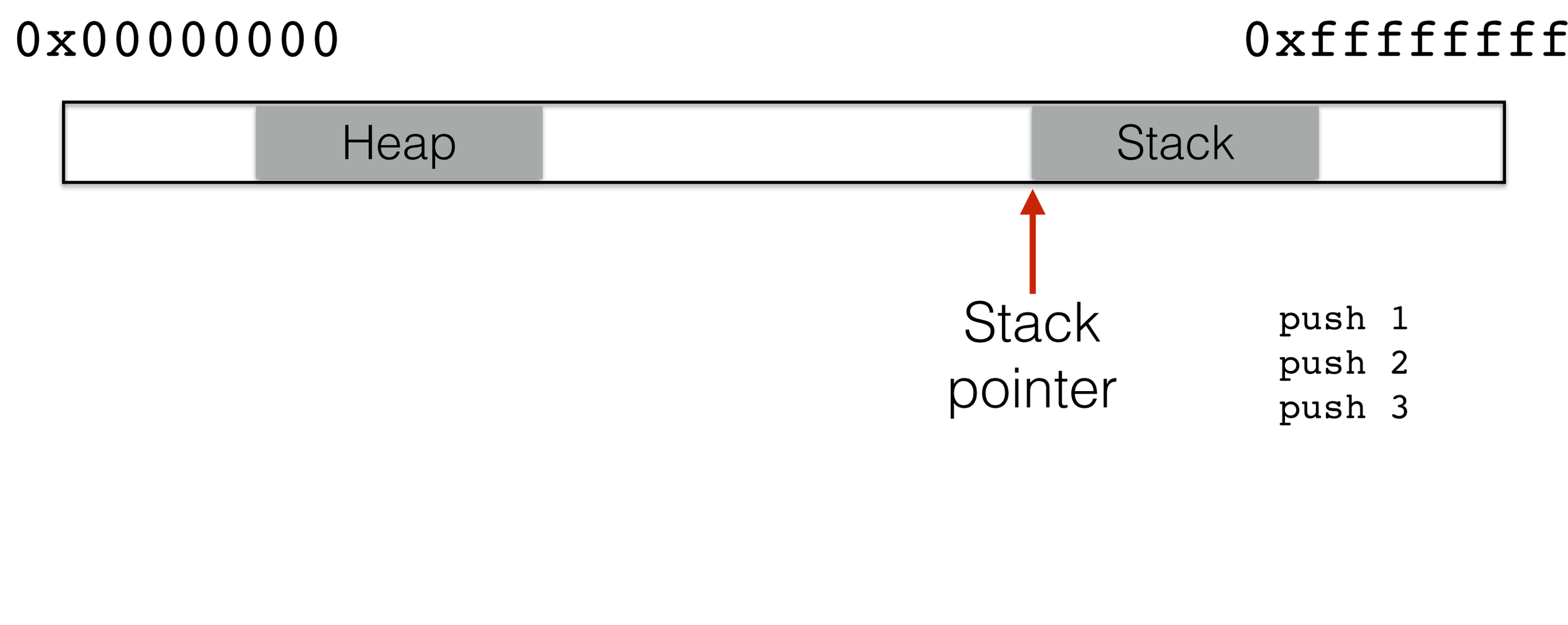
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

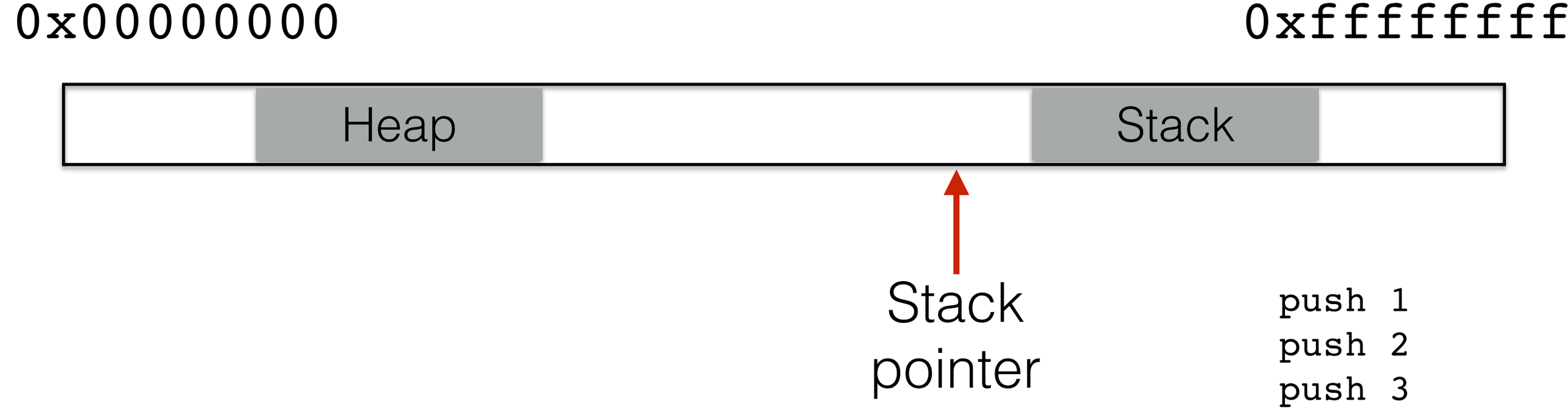
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

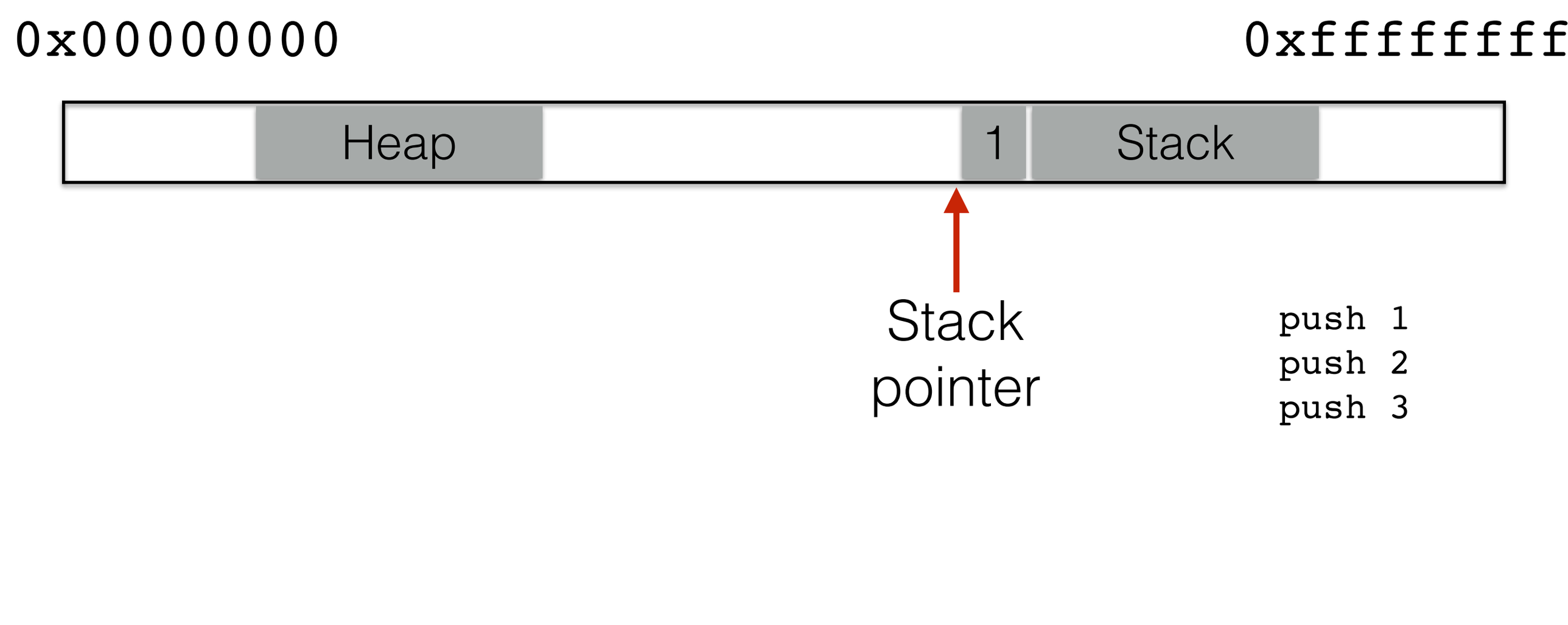
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

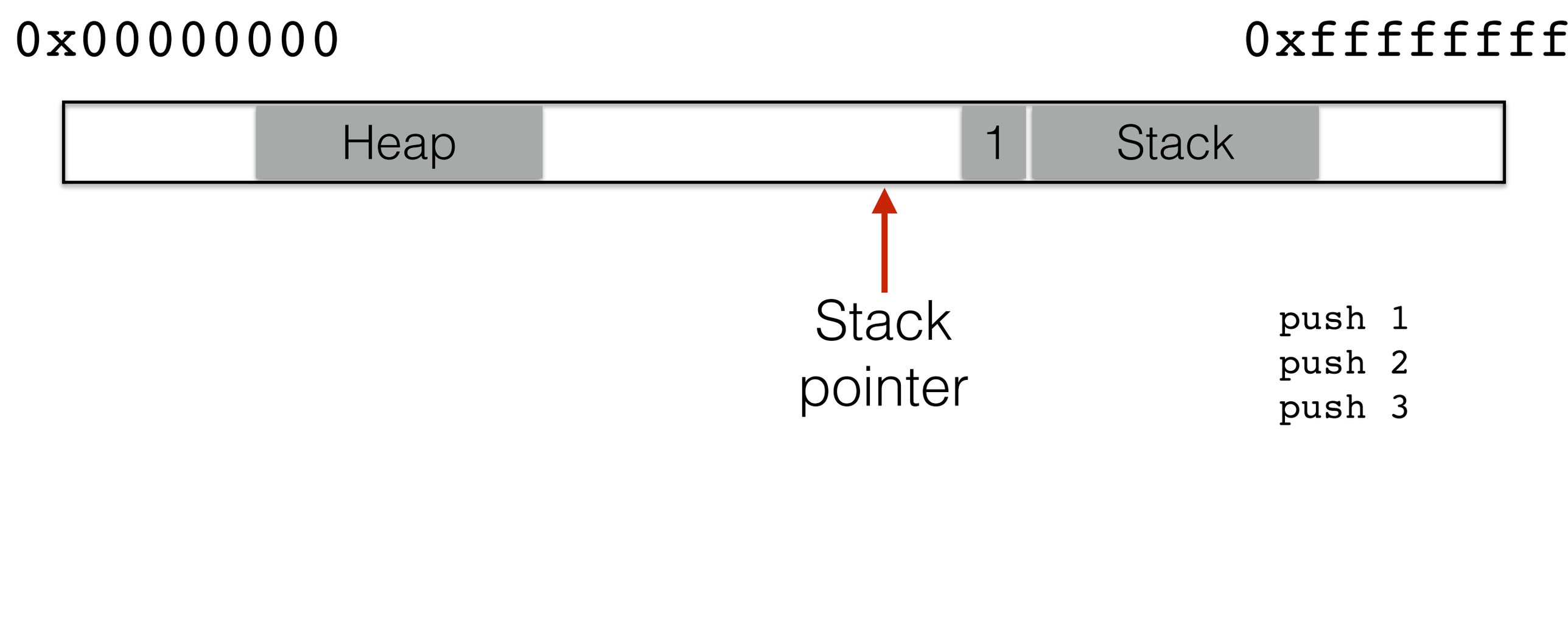
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

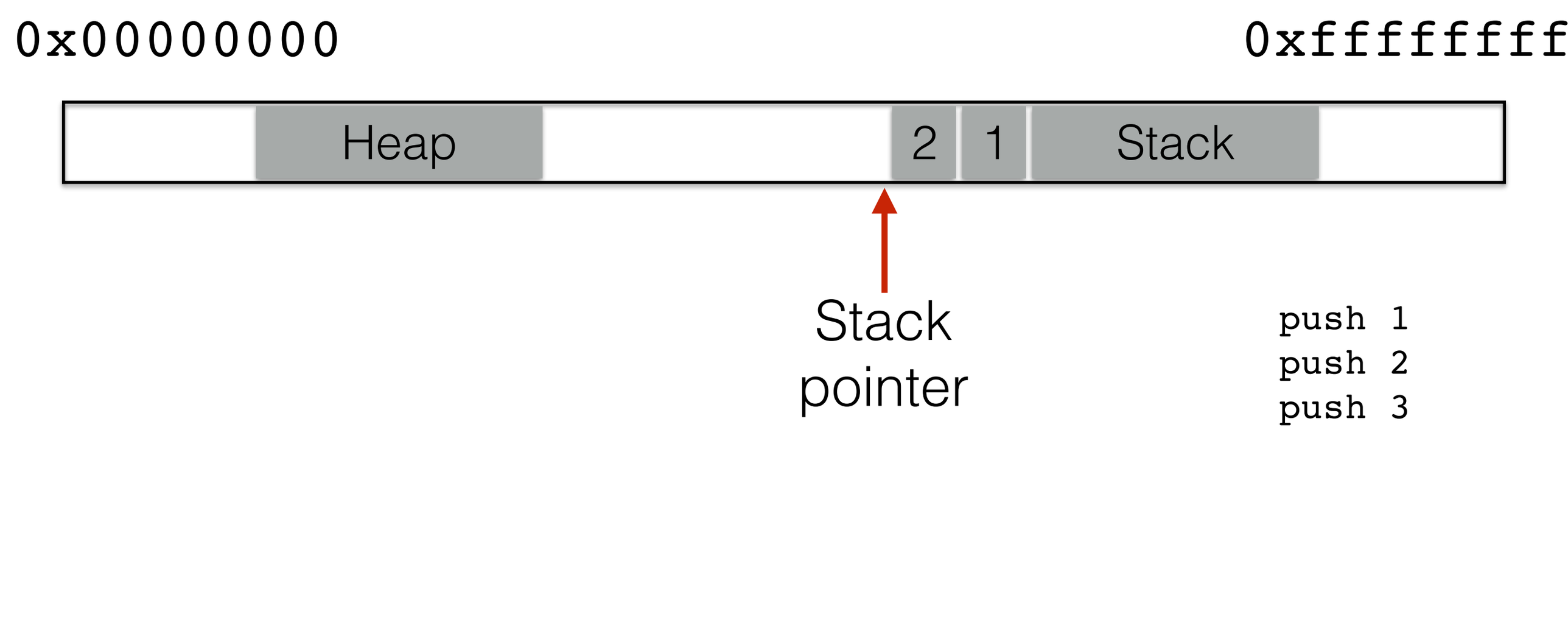
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

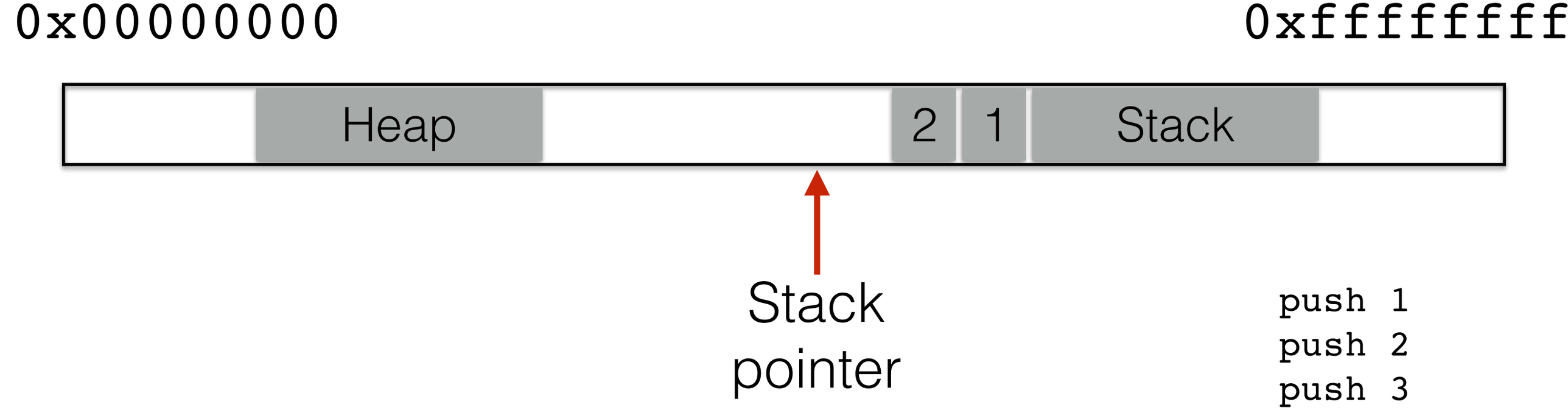
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

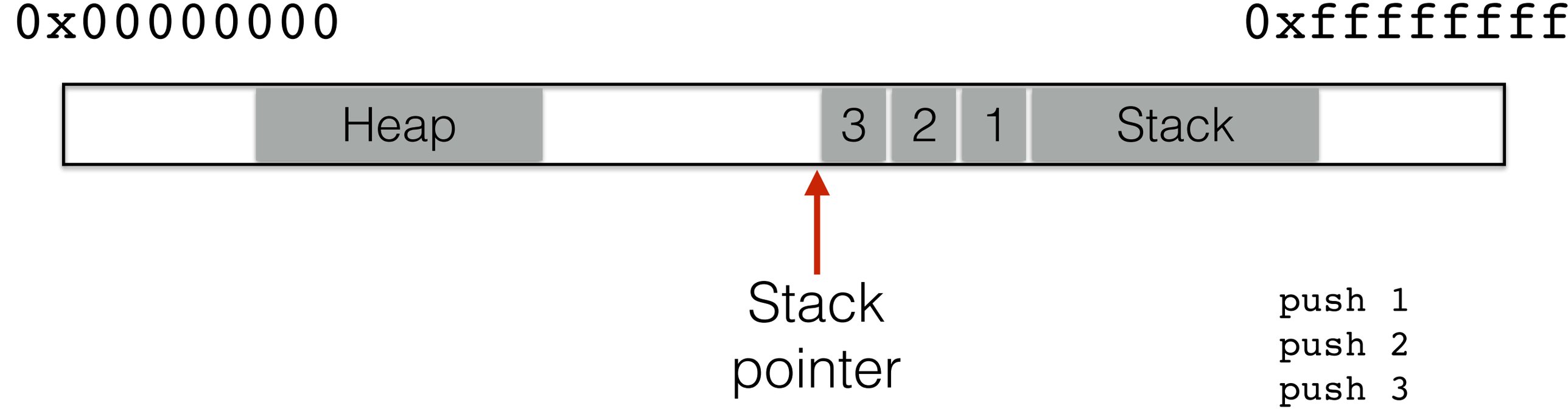
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

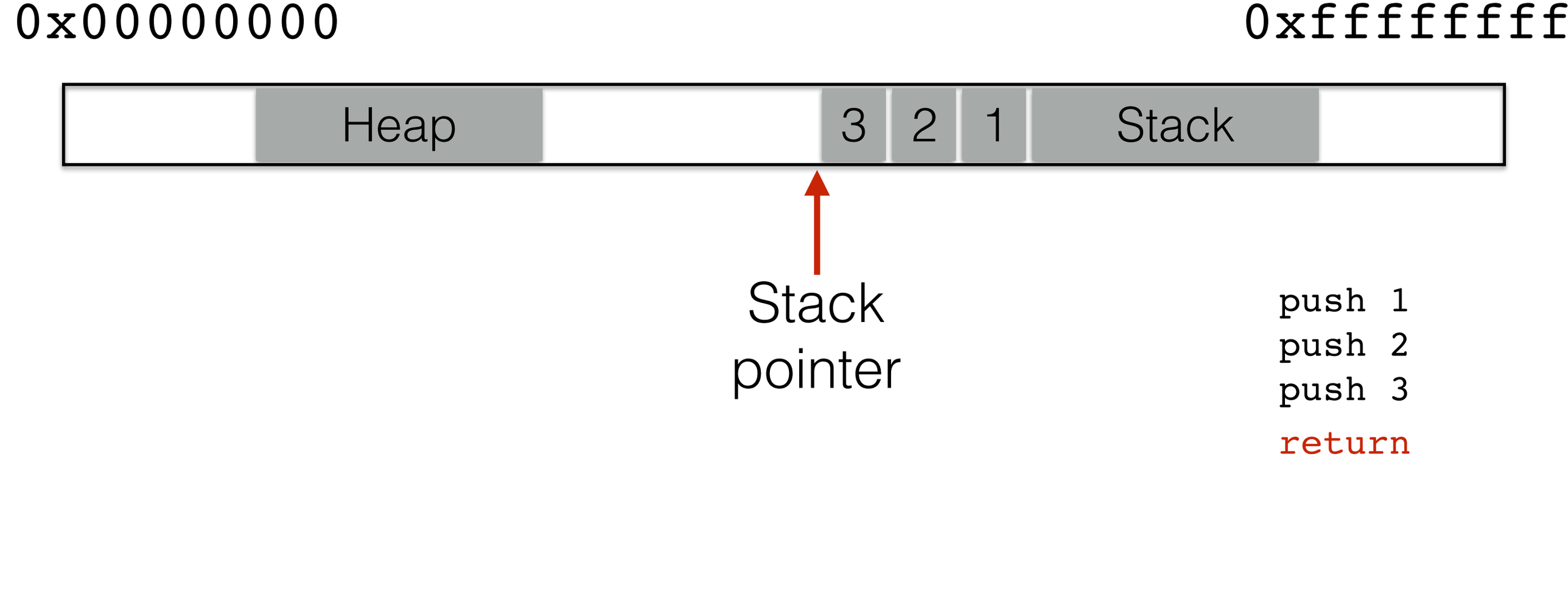
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

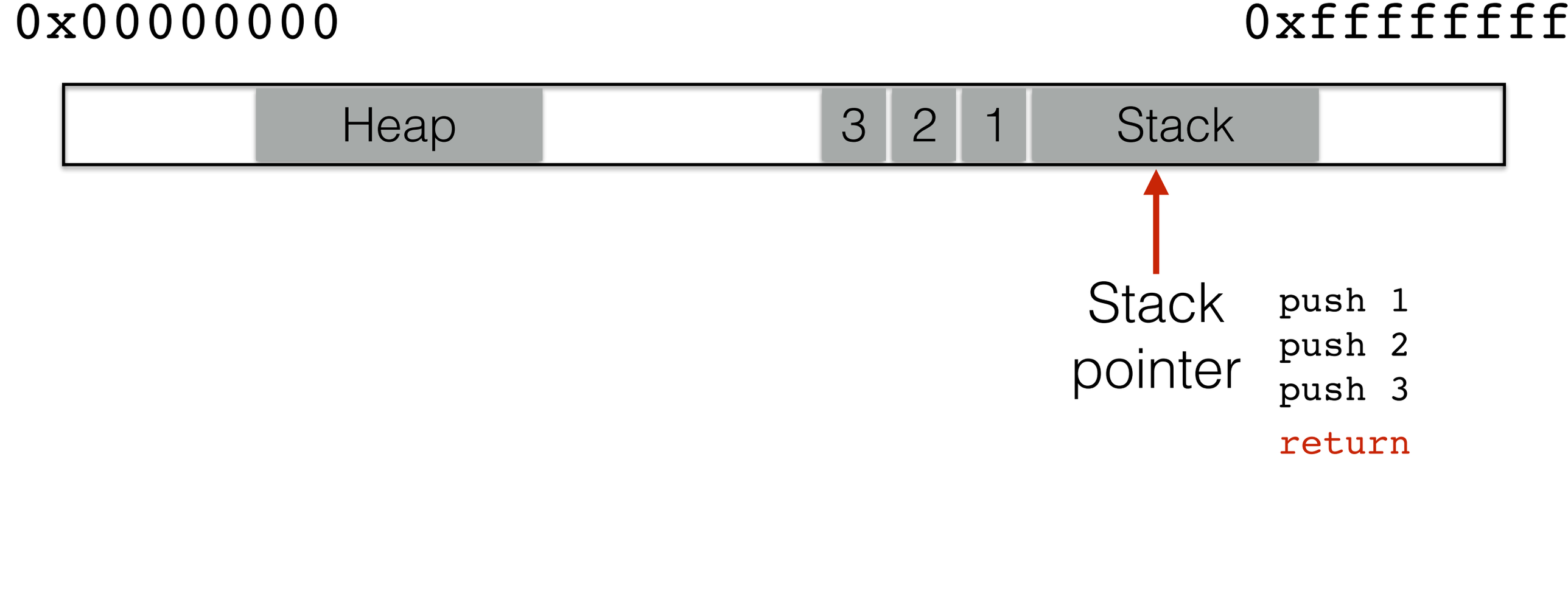
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

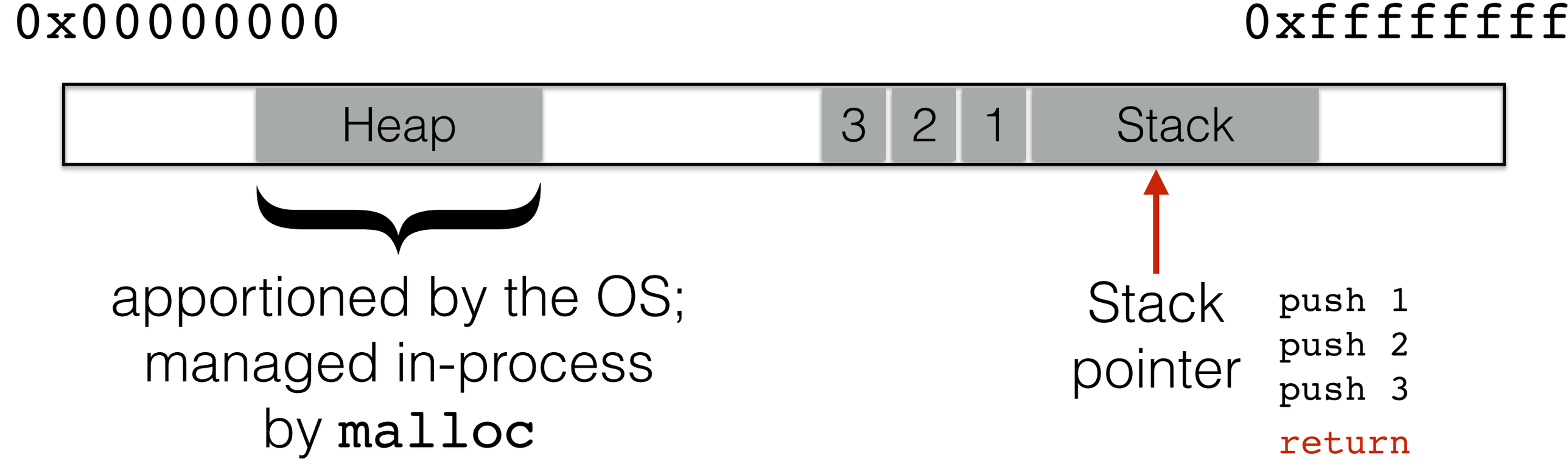
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

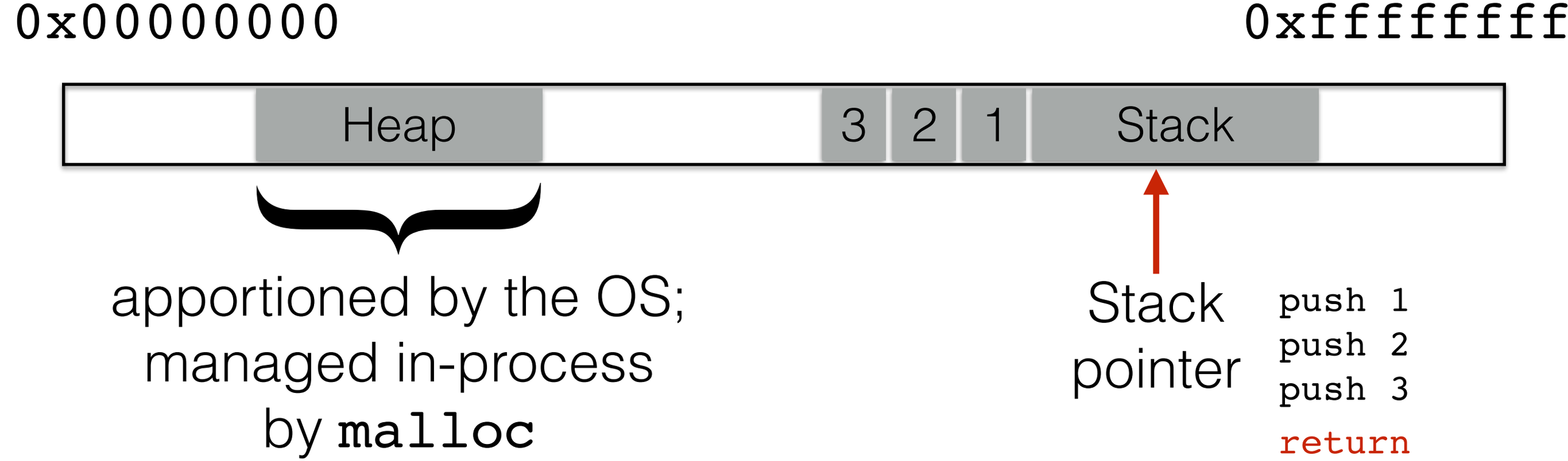
Compiler provides instructions that
adjusts the size of the stack at runtime



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



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)

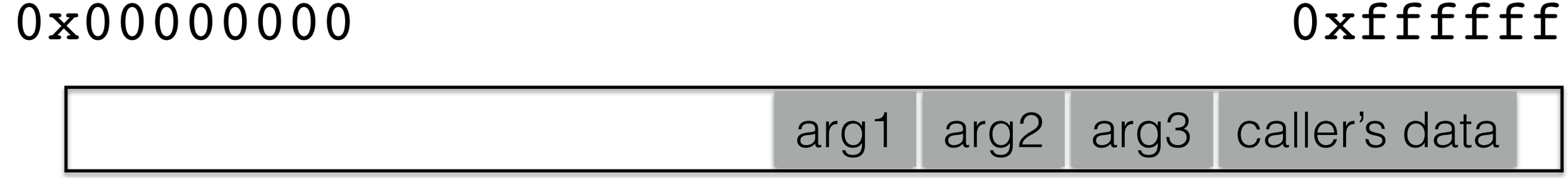
Stack layout when calling functions

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```

0x00000000 0xffffffff

Stack layout when calling functions

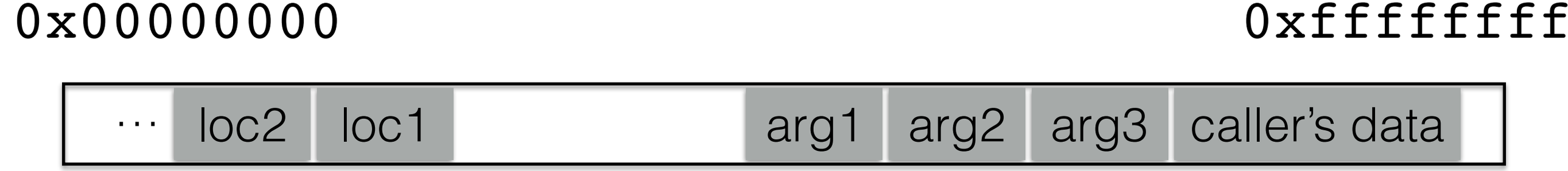
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```



Arguments pushed in reverse order of code

Stack layout when calling functions

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```



Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Stack layout when calling functions

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```

Local variables pushed in the same order as they appear in the code

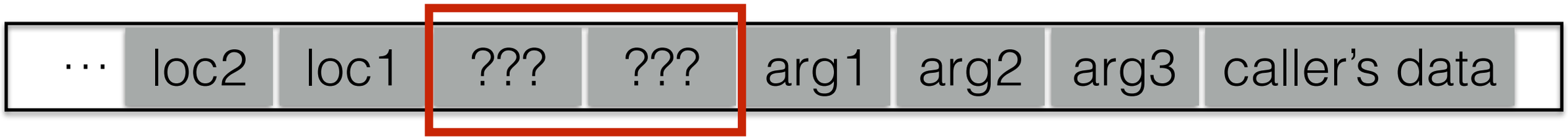
Arguments pushed in reverse order of code

Stack layout when calling functions

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```

Two values between the arguments and the local variables

0x00000000 0xffffffff



Local variables pushed in the same order as they appear in the code

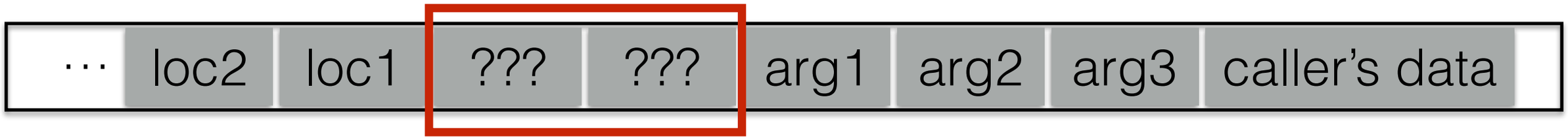
Arguments pushed in reverse order of code

Stack layout when calling functions

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```

Two values between the arguments and the local variables

0x00000000 0xffffffff



Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

We will explore (and attack and defend) them *next time*.