CMSC414 Computer and Network Security

Midterm 1 Recap

Yizheng Chen | University of Maryland
surrealyz.github.io

Mar 5, 2026

Security Principles

- Confidentiality, Integrity, Availability, Authenticity, Authentication
- Security is economics
- Detect if you can’t prevent
- Defense in depth
- Least privilege
- Separation of responsibility / privileges
- Ensure complete mediation
- Don’t rely on security through obscurity
- Use fail-safe defaults
- Design in security from the start
- Consider human factors

Exercise

Security is Economics

- Cost/benefit analyses: The expected benefit to the attacker should ideally be smaller than the expected cost of attack
  - More security (usually) costs more
  - If the attack costs more than the reward, the attacker probably won’t do it
- Corollary: you should focus your energy on securing the weakest links
  - A system is only as secure as the weakest link

Use Fail-Safe Defaults

- Choose default settings that “fail safe,” balancing security with usability when a system goes down
  - e.g., Content Security Policy: By default, reject JavaScript from all websites, use an allowlist to accept some JavaScript from trustworthy website

Principle of Least Privilege

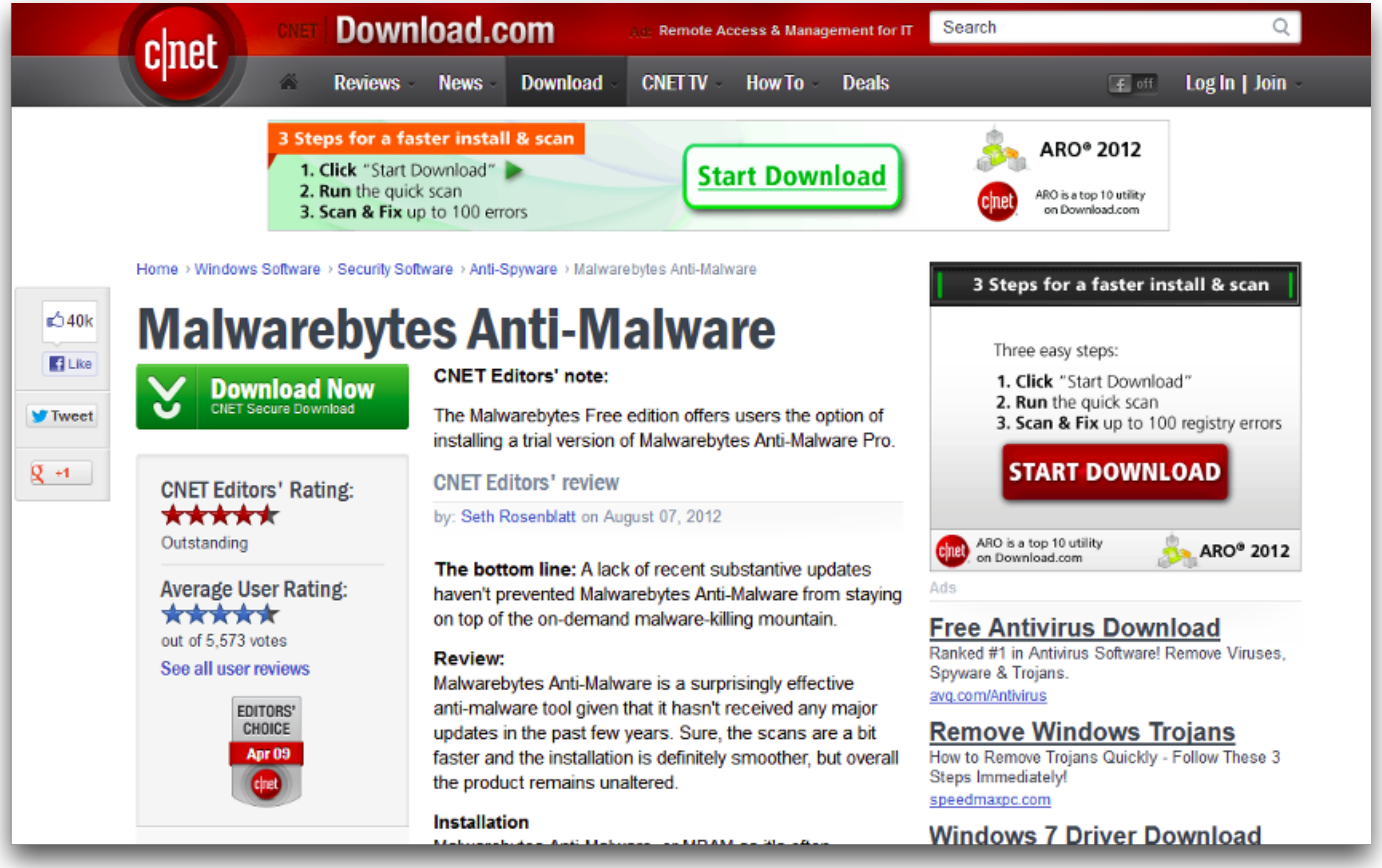
- Consider what permissions an entity or program *needs* to be able to do its job correctly
  - One should only have as much privilege as it *needs*
  - If you grant unnecessary permissions, a malicious or hacked program could use those permissions against you
  - e.g., non-executable pages, same-origin policy

Consider Human Factors

- Users like convenience; if a security system is unusable and not user-friendly, no matter how secure it is, it will go unused
- Example:
  - Pop-up box: install secure update? Users click “remind me later”
  - Automatically downloads important updates by default, easy install and restart
- Consider factors such as developers make mistakes, users are susceptible to social engineering attacks…

Exercise

Clickjacking: Download Buttons



3 Steps for a faster install & scan

1. Click "Start Download"
2. Run the quick scan
3. Scan & Fix up to 100 errors

Home › Windows Software › Security Software › Anti-Spyware › Malwarebytes Anti-Malware

Malwarebytes Anti-Malware

CNET Editors' Rating: ★★★★★ Outstanding

Average User Rating: ★★★★★ out of 5,573 votes

See all user reviews

CNET Editors' note:

The Malwarebytes Free edition offers users the option of installing a trial version of Malwarebytes Anti-Malware Pro.

CNET Editors' review

by: Seth Rosenblatt on August 07, 2012

The bottom line: A lack of recent substantive updates haven't prevented Malwarebytes Anti-Malware from staying on top of the on-demand malware-killing mountain.

Review:
Malwarebytes Anti-Malware is a surprisingly effective anti-malware tool given that it hasn't received any major updates in the past few years. Sure, the scans are a bit faster and the installation is definitely smoother, but overall the product remains unaltered.

Installation

3 Steps for a faster install & scan

Three easy steps:

1. Click "Start Download"
2. Run the quick scan
3. Scan & Fix up to 100 registry errors

Ads

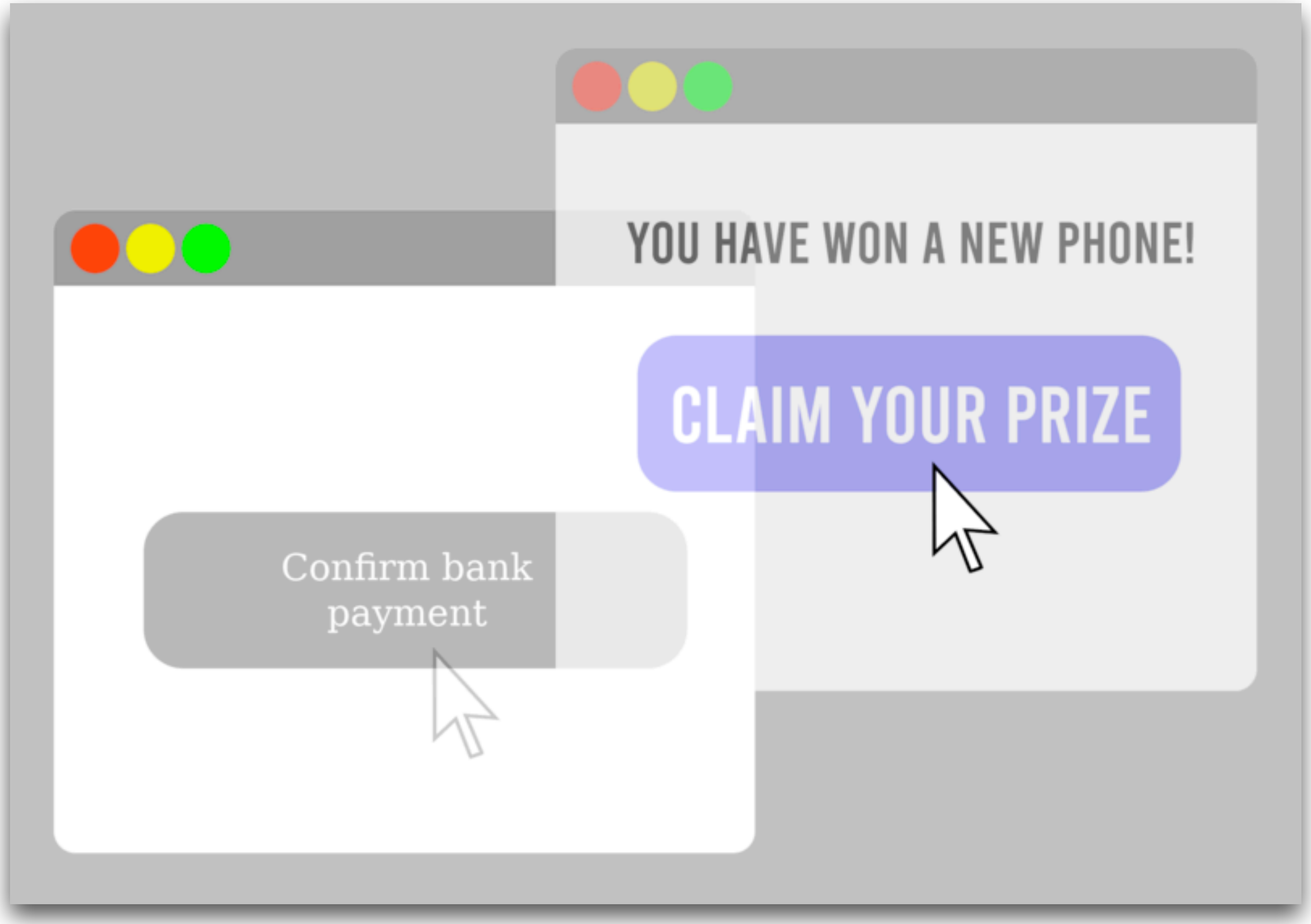
Free Antivirus Download
Ranked #1 in Antivirus Software! Remove Viruses, Spyware & Trojans.
avg.com/Antivirus

Remove Windows Trojans
How to Remove Trojans Quickly - Follow These 3 Steps Immediately!
speedmaxpc.com

Windows 7 Driver Download

- Which is the real download button?
- What if the user clicks the wrong one?

Invisible iframe Variant #1



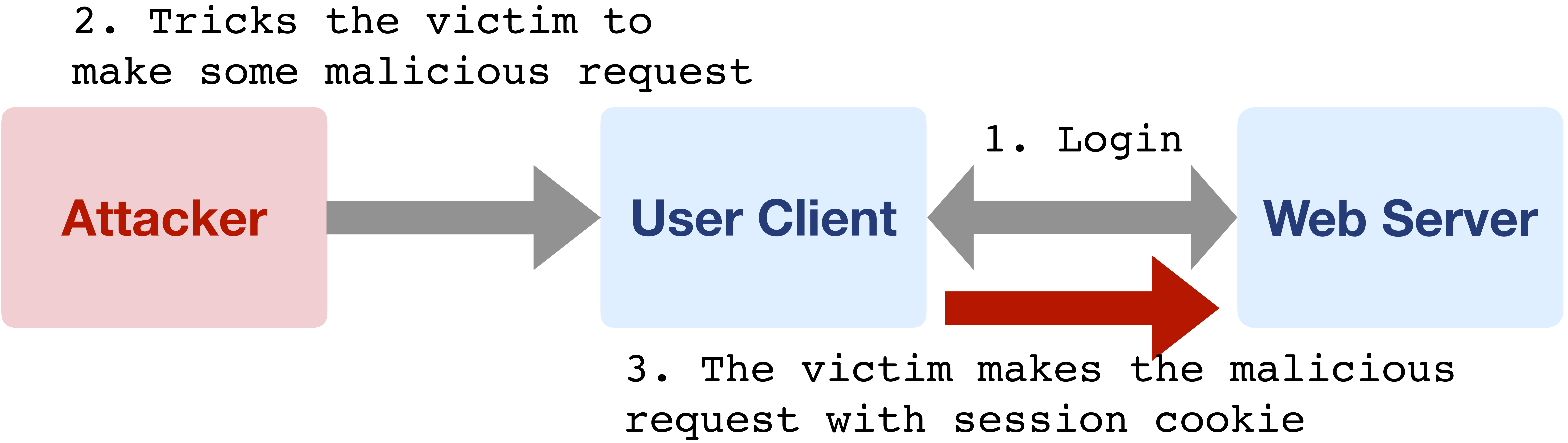
- Frame the legitimate site invisibly, over visible, enticing content
- Victims think they are clicking on the enticing site, but they click on the legitimate site, e.g., pay the attacker's account

Cross-Site Request Forgery (CSRF)

- Idea: What if the attacker tricks the victim into making an unintended request?
  - The victim's browser will automatically attach relevant cookies
  - **The server will think the request came from the victim!**
- **Cross-site request forgery (CSRF or XSRF)**: An attack that exploits cookie-based authentication to perform an action as the victim

Steps of a CSRF Attack

1. User authenticates to the server, receives a **cookie** with a valid **session token**
2. Attacker **tricks** the victim into making a malicious request to the server
3. The victim **makes the malicious request**, attaching the cookie, server accepts it


2. Tricks the victim to make some malicious request
Attacker
User Client
1. Login
Web Server
3. The victim makes the malicious request with session cookie

Cross-Site Scripting (XSS)

- **Cross-site scripting (XSS)**: Injecting JavaScript into websites that are viewed by other users
  - Cross-site scripting subverts the same-origin policy
- Two main types of XSS
  - Stored XSS
  - Reflected XSS

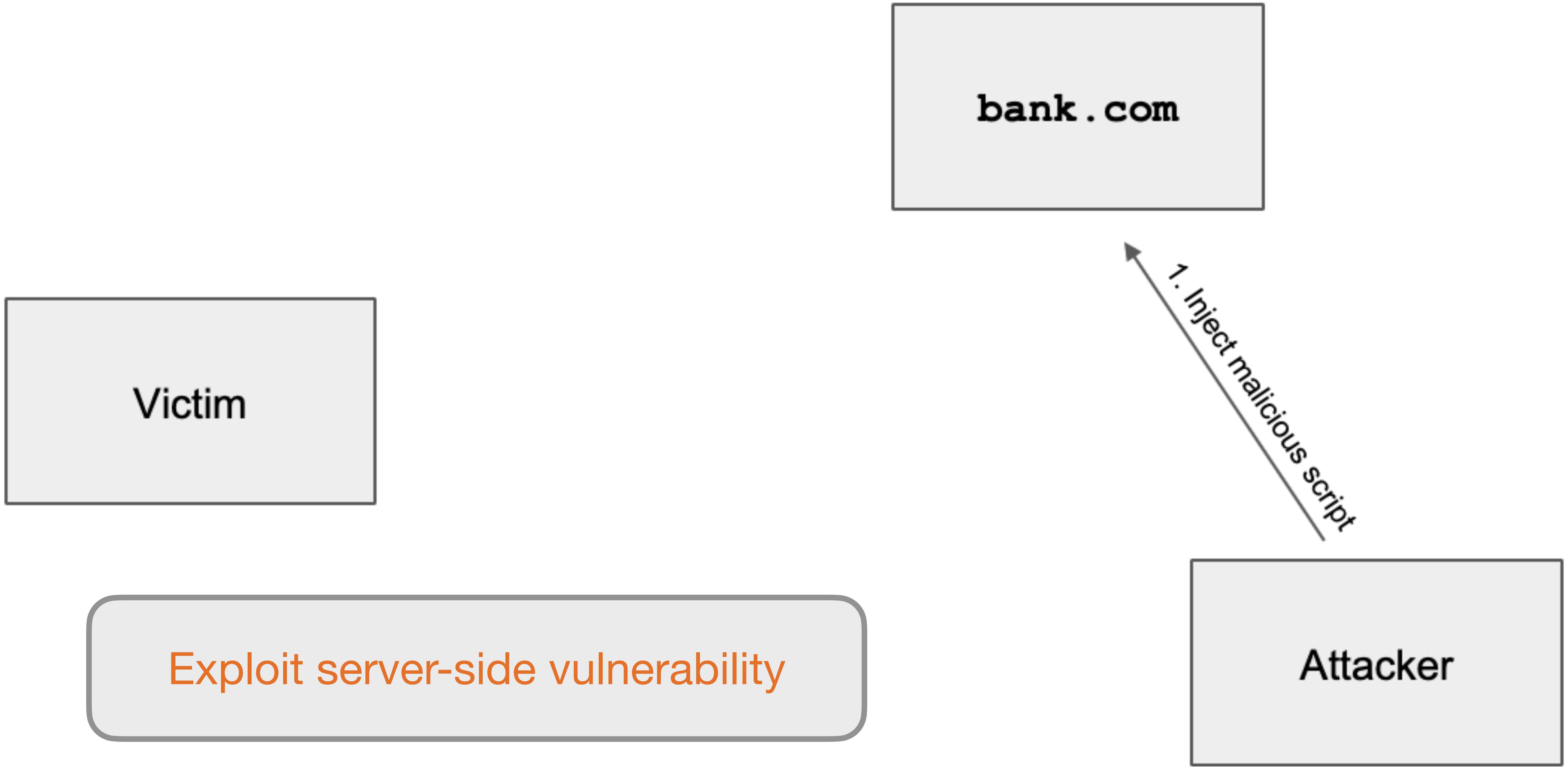
Stored XSS

- **Stored XSS (persistent XSS):** The attacker's JavaScript is stored on the legitimate server and sent to browsers
- Classic example: Facebook pages
  - An attacker puts some JavaScript on their Facebook page
  - Anybody who loads the attacker's page will see JavaScript (with the origin of Facebook)

Stored XSS

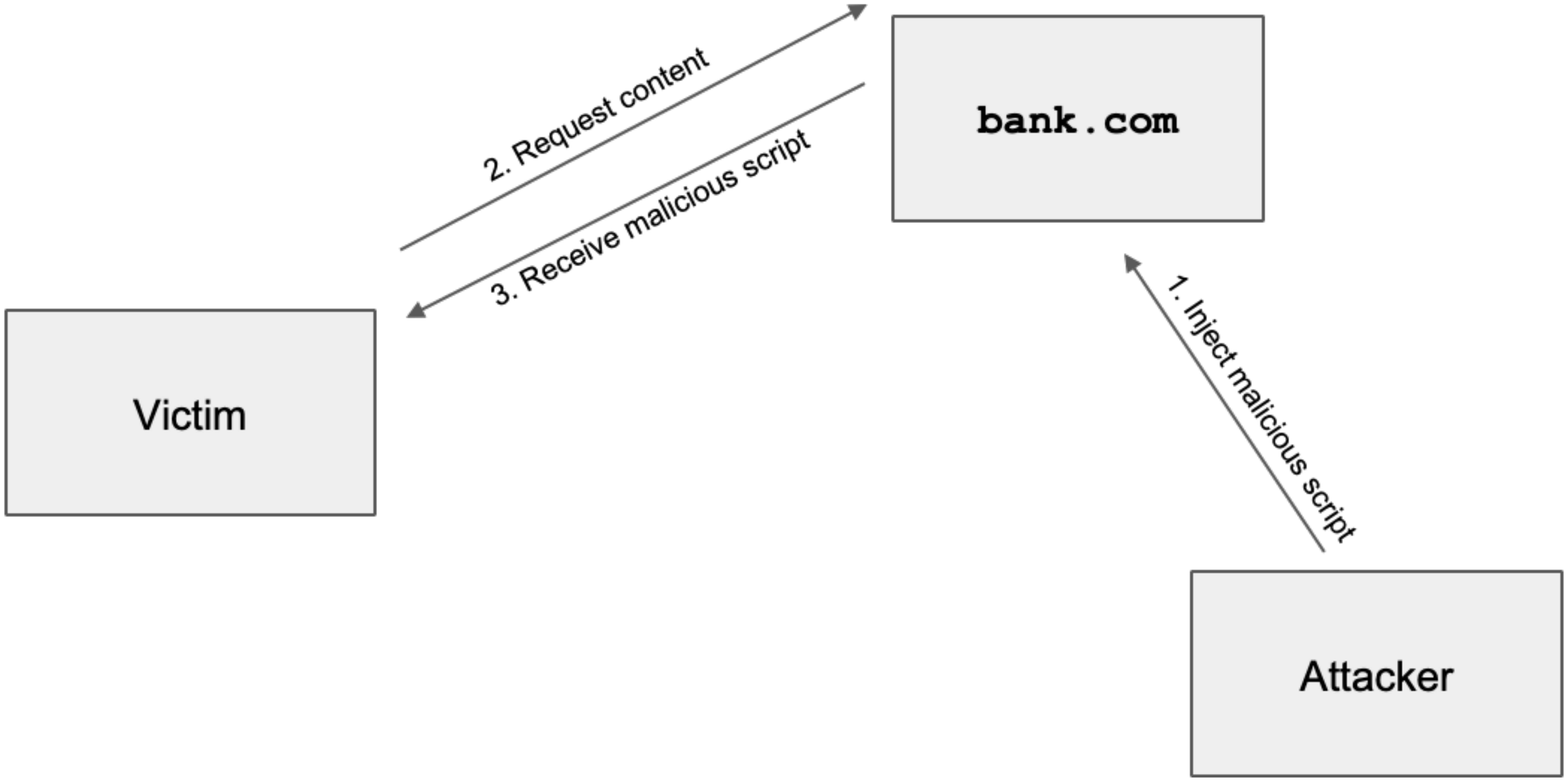
- **Stored XSS (persistent XSS)**: The attacker's JavaScript is stored on the legitimate server and sent to browsers
- Classic example: Facebook pages
  - An attacker puts some JavaScript on their Facebook page
  - Anybody who loads the attacker's page will see JavaScript (with the origin of Facebook)
- Stored XSS requires the victim to load the page with injected JavaScript
- Remember: Stored XSS is a **server-side vulnerability**!

Stored XSS

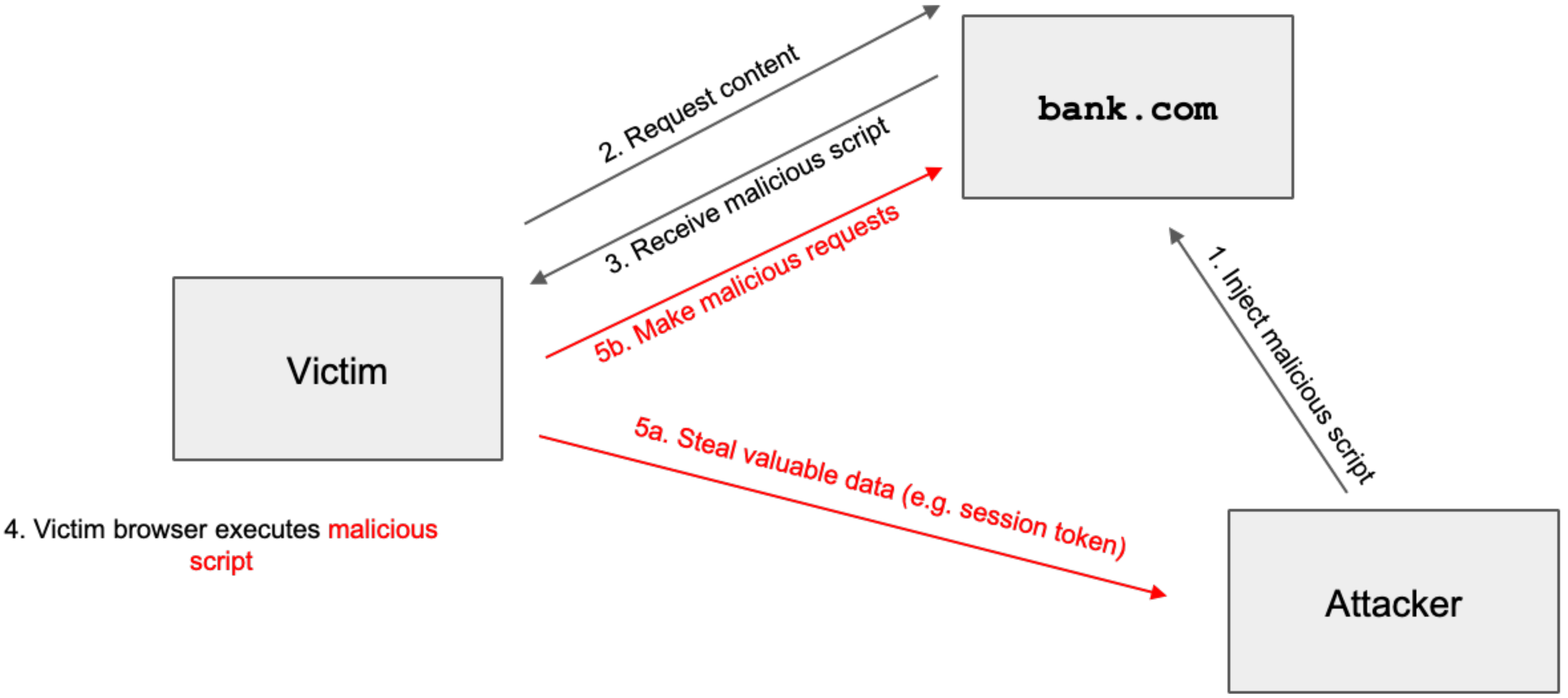


Exploit server-side vulnerability

Stored XSS


bank.com
Victim
Attacker
1. Inject malicious script
2. Request content
3. Receive malicious script

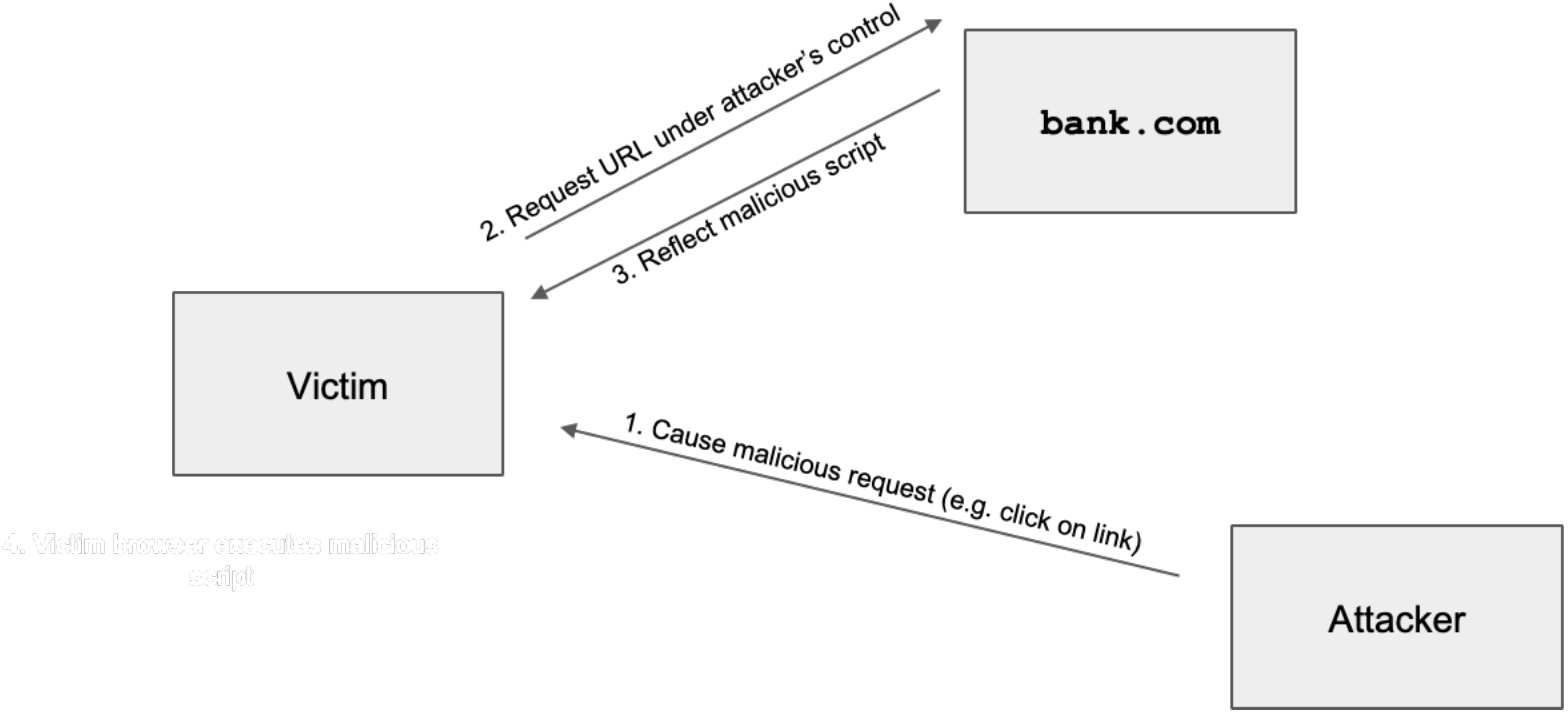
Stored XSS


bank.com
Victim
Attacker
1. Inject malicious script
2. Request content
3. Receive malicious script
4. Victim browser executes malicious script
5a. Steal valuable data (e.g. session token)
5b. Make malicious requests

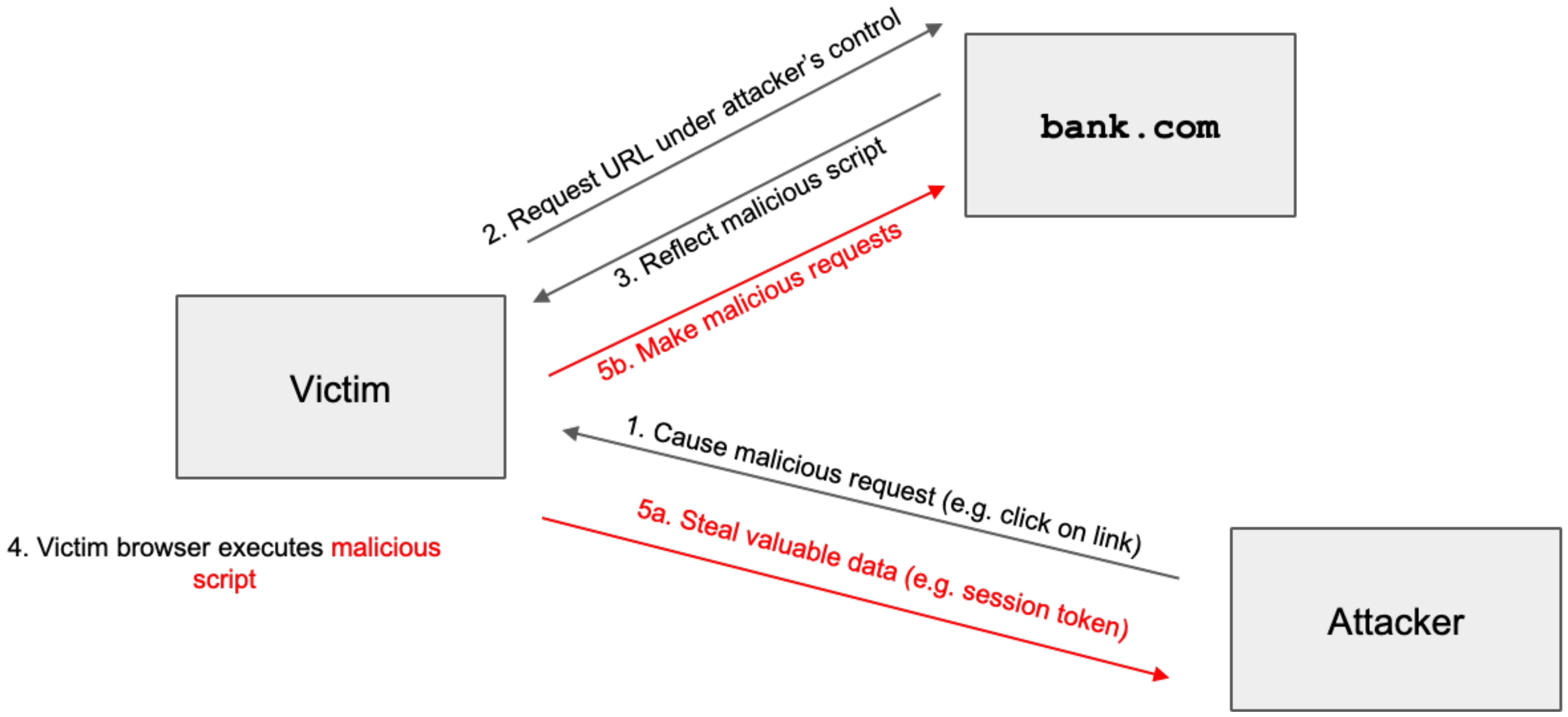
Reflected XSS

- **Reflected XSS:** The attacker causes the victim to input JavaScript into a request, and the content is **reflected (copied)** in the response from the server
    - Classic example: Search
    - If you make a request to http://google.com/search?q=bot, the response will say “10,000 results for bot”
    - If you make a request to http://google.com/search?q=<script>alert(1)</script>, the response will say “10,000 results for <script>alert(1)</script>”
- Reflected XSS requires the victim to make a request with injected JavaScript

Reflected XSS


bank.com
2. Request URL under attacker's control
3. Reflect malicious script
Victim
1. Cause malicious request (e.g. click on link)
Attacker
4. Victim browser executes malicious script

Reflected XSS


bank.com
Victim
Attacker
2. Request URL under attacker's control
3. Reflect malicious script
5b. Make malicious requests
1. Cause malicious request (e.g. click on link)
5a. Steal valuable data (e.g. session token)
4. Victim browser executes malicious script

Reflected XSS: Making a Request

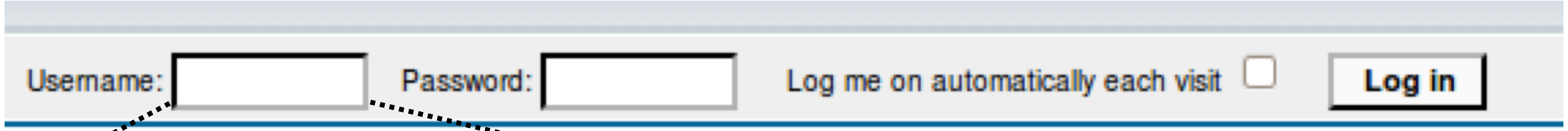
- How do we force the victim to make a request to the legitimate website with injected JavaScript?
  - Trick the victim into visiting the attacker's website, and include an embedded iframe that makes the request
    - Can make the iframe very small (1 pixel x 1 pixel), so the victim doesn't notice it:

      ```
      <iframe height=1 width=1 src="http://google.com/search?
      q=<script>alert(1)</script>">
      ```
  - clicking a link (e.g. posting on social media, sending a text, etc.)
  - visiting the attacker's website, which redirects to the reflected XSS link
  - …

Reflected XSS is not CSRF

- Reflected XSS and CSRF both require the victim to make a request to a link
- Reflected XSS: An HTTP response contains maliciously inserted JavaScript, executed on the client side
- CSRF: A malicious HTTP request is made (containing the user's cookies), executing an effect on the server side

SQL injection



```
frank’ OR 1=1); --
```

```
$result = mysql_query(“select * from Users
       where(name=‘$user’ and password=‘$pass’);”);
```

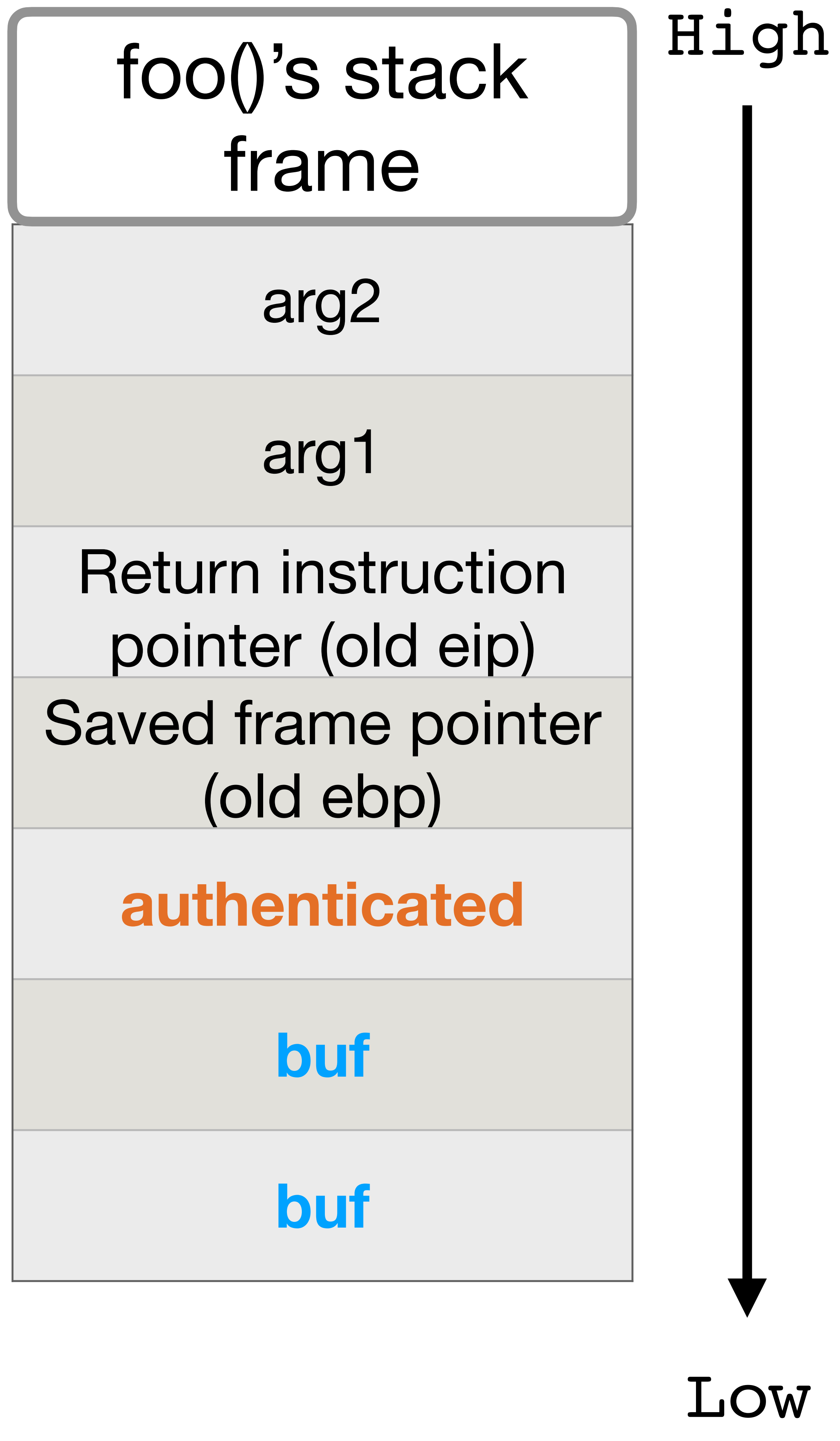
```
$result = mysql_query(“select * from Users where
(name=‘frank’ OR 1=1); -- ‘ and password=‘x’);”);
```

Exercise

How to change authenticated to 1?

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1, int arg2) {
    int  authenticated = 0;
    char buf[8];
    ...
}
```



foo()'s stack frame
arg2
arg1
Return instruction pointer (old eip)
Saved frame pointer (old ebp)
authenticated
buf
buf
High
Low

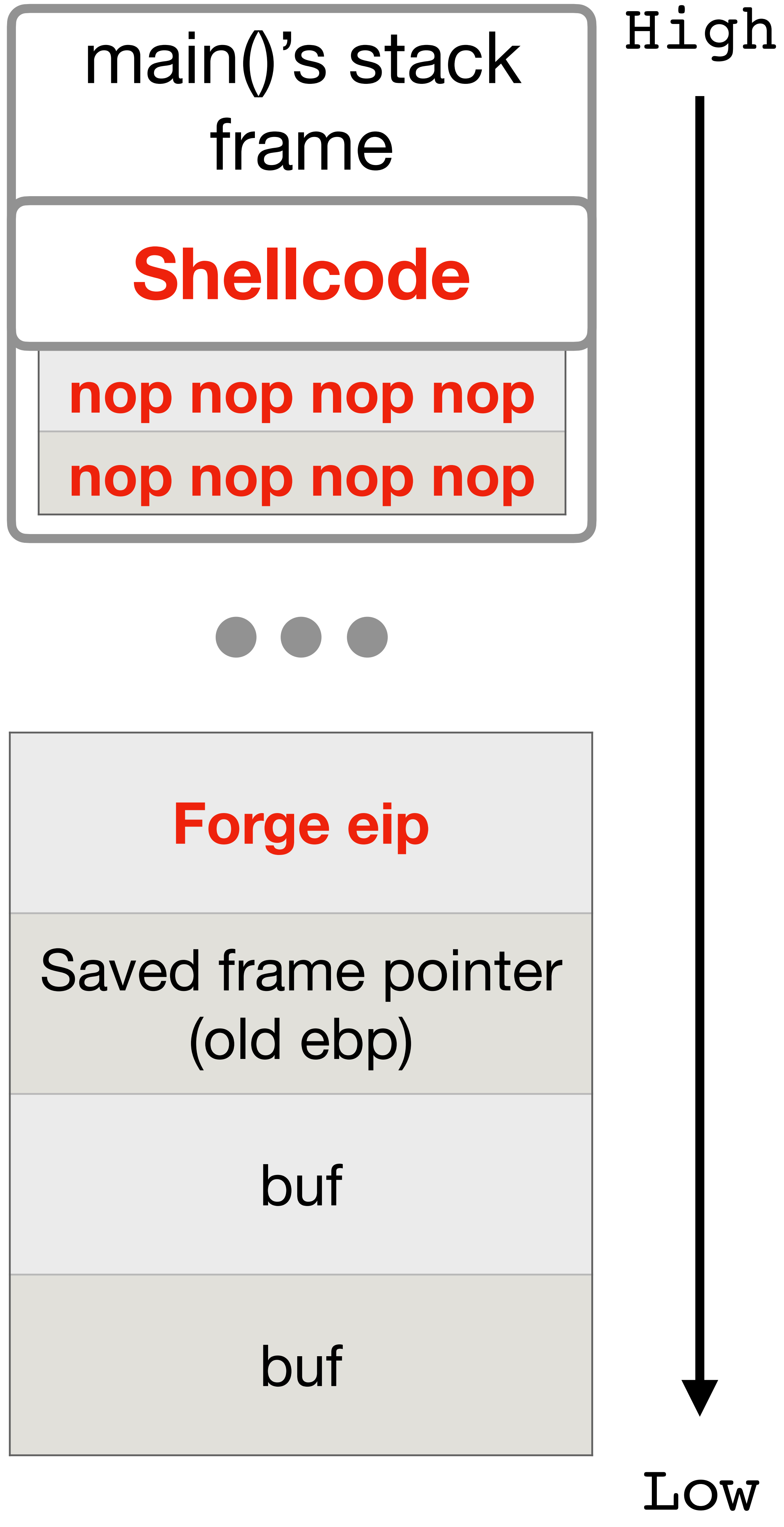
NOP

- nop is a single-byte instruction (just moves to the next instruction)
- 0x90

NOP Sled

```
void main() {
    vulnerable();
}

void
vulnerable(?????????) {
    char buf[8];
    gets(buf)
    ...
}
```



main()'s stack frame

Shellcode

nop nop nop nop

nop nop nop nop

Forge eip

Saved frame pointer (old ebp)

buf

buf

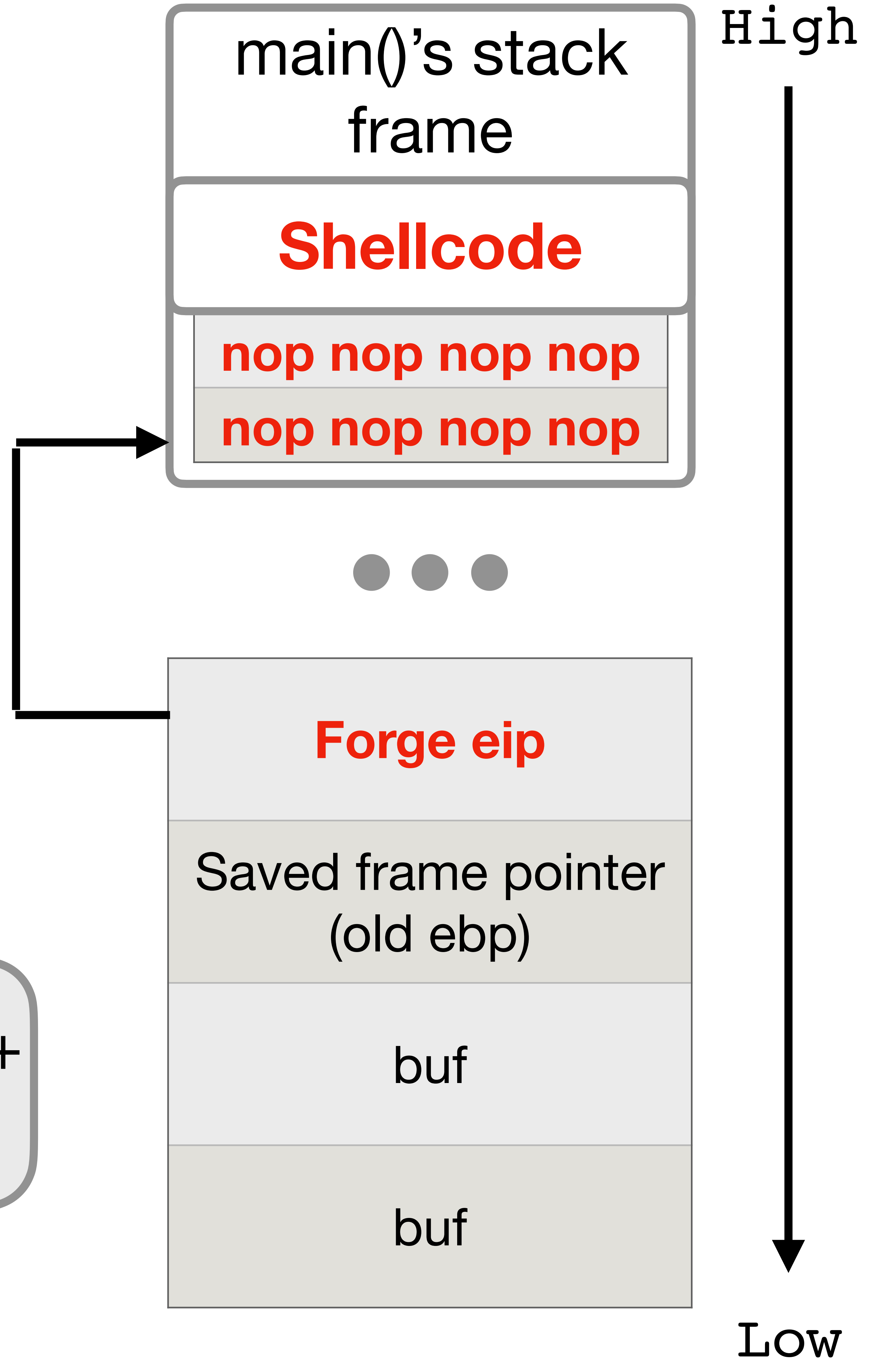
High

Low

NOP Sled

```
void main() {
    vulnerable();
}

void
vulnerable(?????????) {
    char buf[8];
    gets(buf)
    ...
}
```



High

main()'s stack frame

Shellcode

nop nop nop nop

nop nop nop nop

• • •

Forge eip

Saved frame pointer (old ebp)

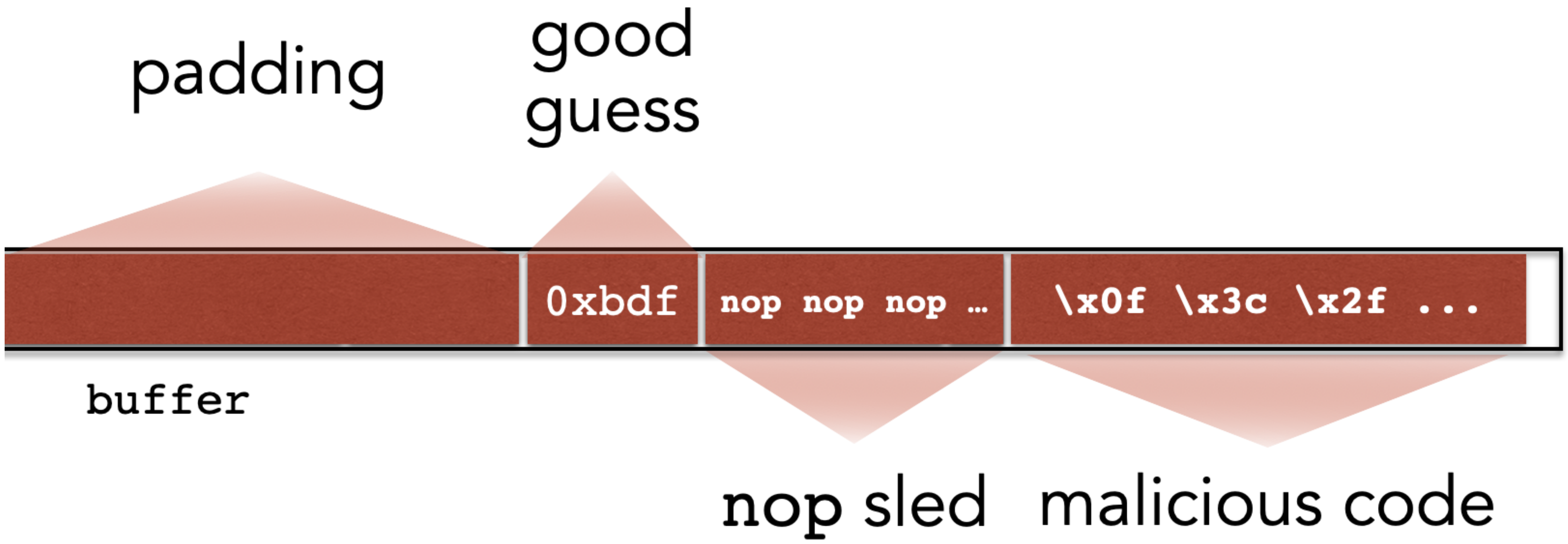
buf

buf

Low

[12 bytes of garbage] + **[guess somewhere in the NOP]** + [a lot of NOPs] + [shellcode]

NOP Slide / NOP Sled



- Putting the shell code in the end of the payload buffer can maximize the number of NOPs
- Good guess of somewhere in NOP: jumping anywhere inside the NOP will make the attack successful.
- **This improves our chances of guessing by a factor of # of NOPs.**

Exercise

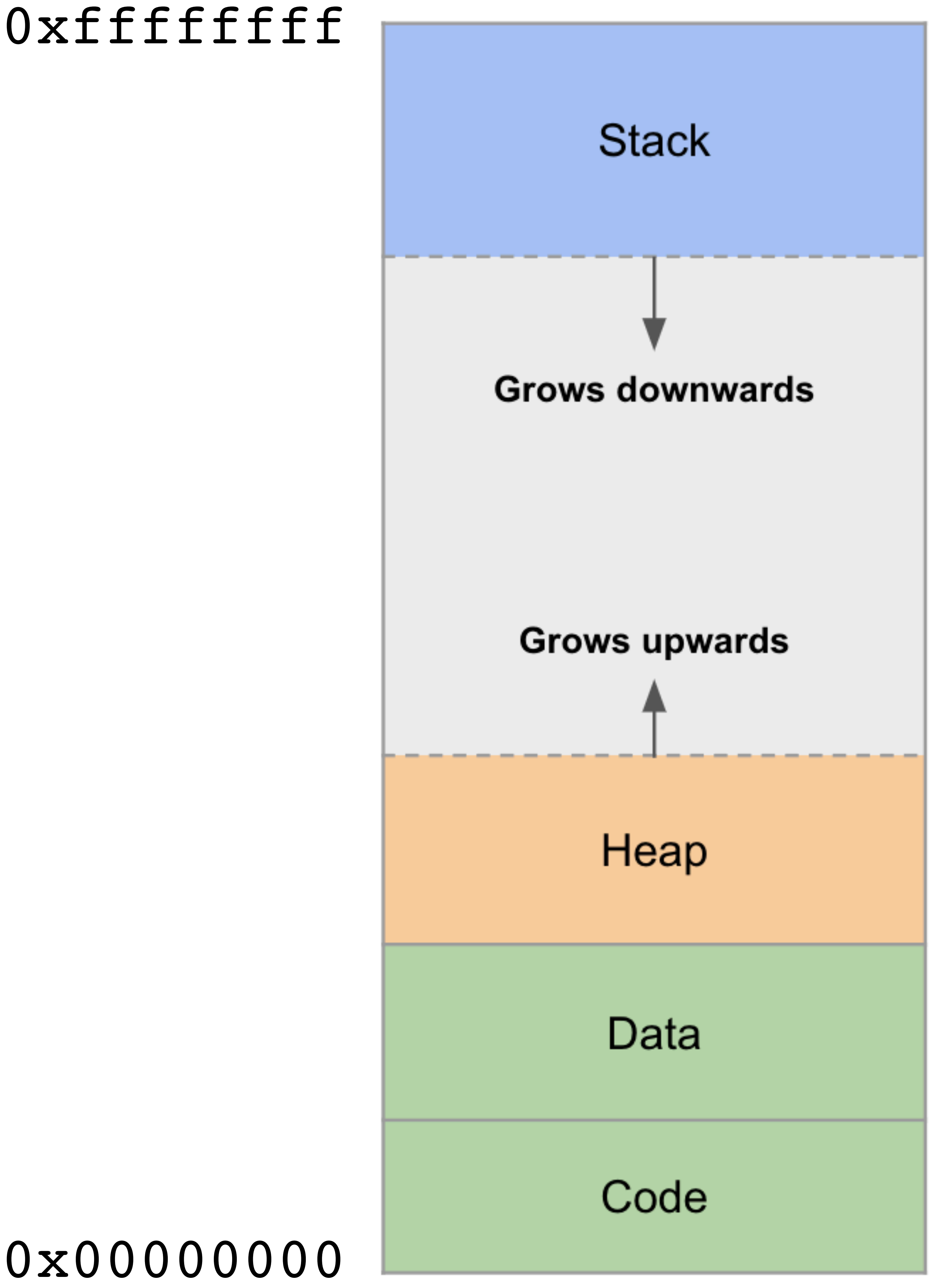
Address Space Layout Randomization

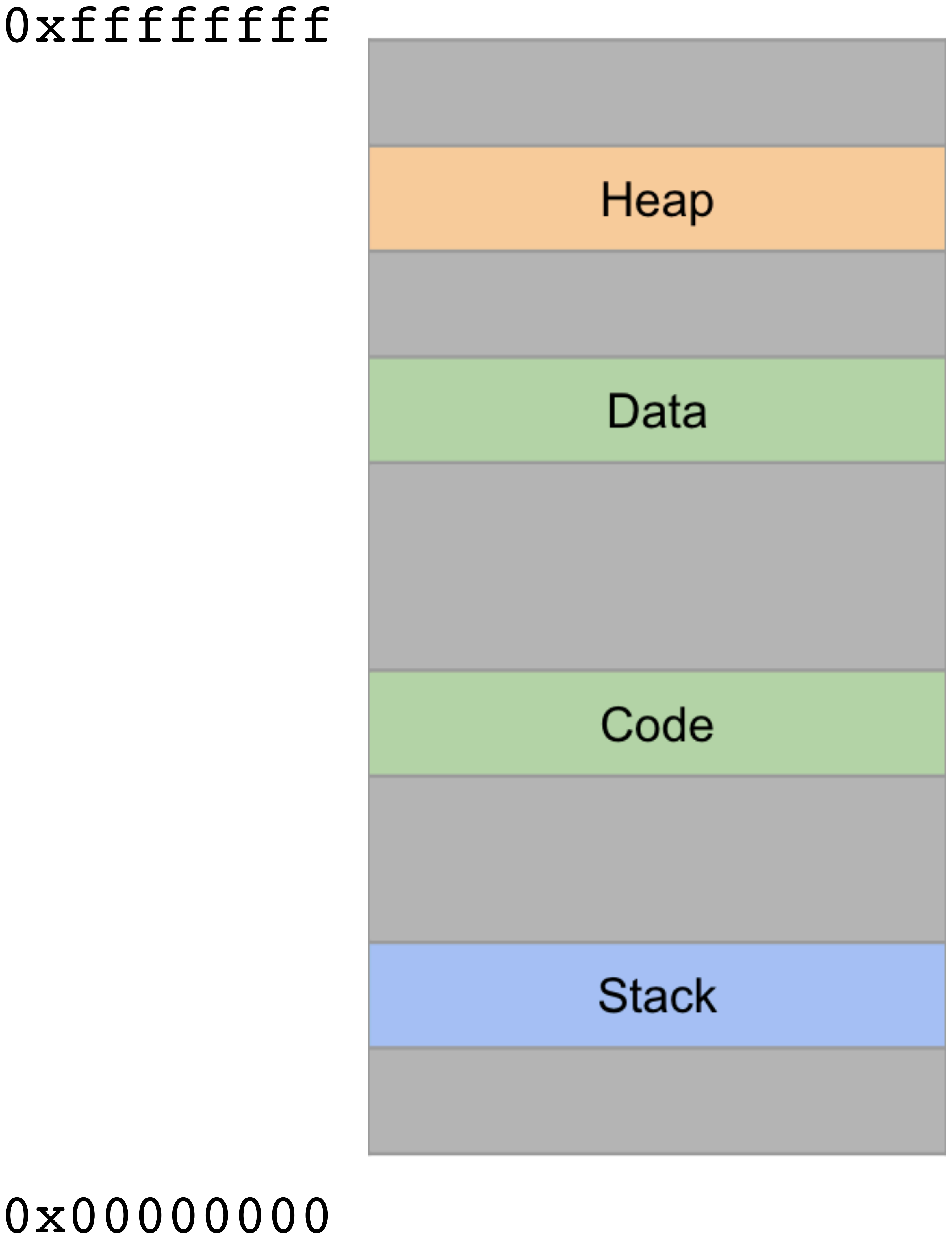
- Goal: make it hard for attackers to place shell code on the stack, on the heap, or find out the address of the code

- Randomize the addresses of code, data, heap, stack

- Theoretically, very hard to know the addresses, so we can mitigate the attacks

Address Space Layout Randomization





Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
    - Programs are dynamically linked at runtime, so ASLR has almost no overhead

Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead
- However…
- Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)
  - Leak the address of a pointer, whose address relative to your shellcode is known (stack pointer, RIP)
  - Guess the address of your shellcode: Brute-force

Exercise Before Android Question

Mitigating Memory Safety Attacks

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode

Mitigating Memory Safety Attacks

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
    - Mitigation: Address Space Layout Randomization (ASLR)
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode

Mitigating Memory Safety Attacks

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode
    - Mitigation: Non-executable pages

Non-Executable Pages

- Idea: Most programs don’t need memory that is both written to and executed, so make portions of memory either executable or writable but not both
  - Stack, heap, and static data: Writable but not executable
  - Code: Executable but not writable
- Also known as
  - W^X (write XOR execute)
  - DEP (Data Execution Prevention, name used by Windows)
  - No-execute bit

Non-Executable Pages

- Security Principles of Non-Executable Pages?
- How to subvert non-executable pages?

Subverting Non-Executable Pages

- **Return-to-libc**: An exploit technique that overwrites the RIP to jump to a function in the standard C library (libc) or a common operating system function
- **Return-oriented programming (ROP)**: Constructing custom shellcode using pieces of code that already exist in memory

Mitigating Memory Safety Attacks

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
    - Mitigation: Stack Canaries
    - Mitigation: Pointer Authentication
4. Return from the function
5. Begin executing malicious shellcode

Combining Mitigations

- **Defense in depth**
- Example: Combining ASLR and non-executable pages
- To defeat ASLR and non-executable pages, the attacker needs to find two vulnerabilities
    - First, find a way to leak memory and reveal the address randomization (defeat ASLR)
    - Second, find a way to write to memory and write a ROP chain (defeat non-executable pages)

Exercise