# Chapter 20: Vulnerability Analysis

- What is a vulnerability?
- Example vulnerabilities
- Penetration studies
  - Flaw Hypothesis Methodology
  - Examples
- Vulnerability classification schemes

# Definitions

- *Vulnerability (security flaw)*: failure of security policies, procedures, and controls that allow a subject to commit an action that violates the security policy
  - Subject is called an *attacker*
  - Using the failure to violate the policy is *exploiting the vulnerability* or *breaking in*

# What is an Exploit?

- An exploit is any input (i.e., a piece of software, an argument string, or sequence of commands) that takes advantage of a bug, glitch or vulnerability in order to cause an attack
- An attack is an unintended or unanticipated behavior that occurs on computer software, hardware, or something electronic and that brings an advantage to the attacker

# Incomplete List of Vulnerabilities – Latest Version

- Buffer overflows/memory safety errors
- Integer overflow, underflow, and sign conversion errors
- Null pointer errors
- Race conditions, atomicity violations, TOCTTOU (time of check to time of use) vulnerabilities
- Resource drains that may be subject to attack (e.g., memory or file handle leaks)
- Other memory errors (double frees, use after free, uninitialized memory use)
- Insufficient access control checks/incomplete mediation
- Tainted data/input validation errors (e.g., format string vulnerabilities, lack of checking on command strings sent to OS, insufficient restrictions on input file names)
- Unhandled exceptions or returned error status codes
- Leaks of confidential information

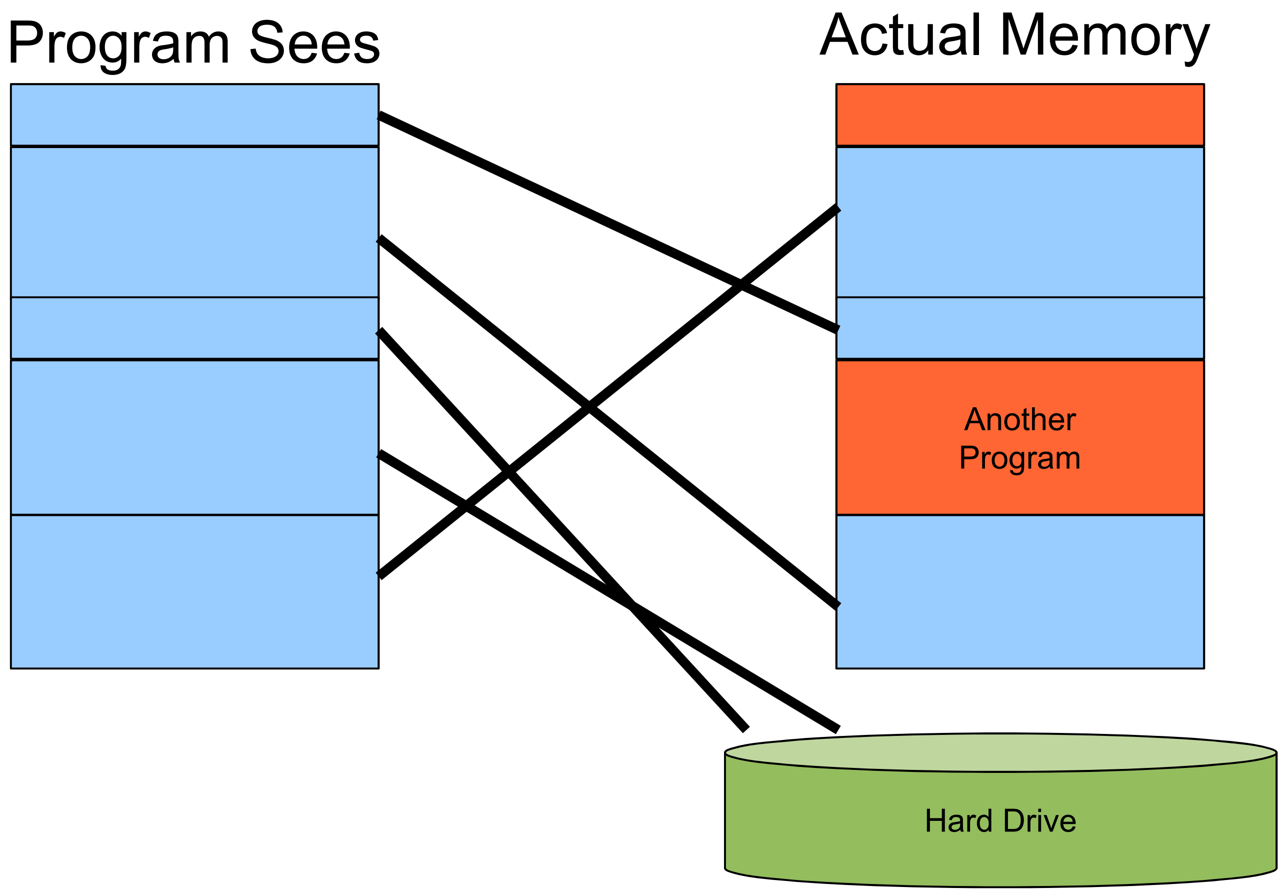
# Buffer Overflow Attack

- One of the most common OS bugs is a buffer overflow
  - The developer fails to include code that checks whether an input string fits into its buffer array
  - An input to the running process exceeds the length of the buffer
  - The input string overwrites a portion of the memory of the process
  - Causes the application to behave improperly and unexpectedly
- Effect of a buffer overflow
  - The process can operate on malicious data or execute malicious code passed in by the attacker
  - If the process is executed as root, the malicious code will be executing with root privileges

# Address Space

- Every program needs to access memory in order to run
- For simplicity sake, it would be nice to allow each process (i.e., each executing program) to act as if it owns all of memory
- The address space model is used to accomplish this
- Each process can allocate space anywhere it wants in memory
- Most kernels manage each process' allocation of memory through the virtual memory model
- How the memory is managed is irrelevant to the process

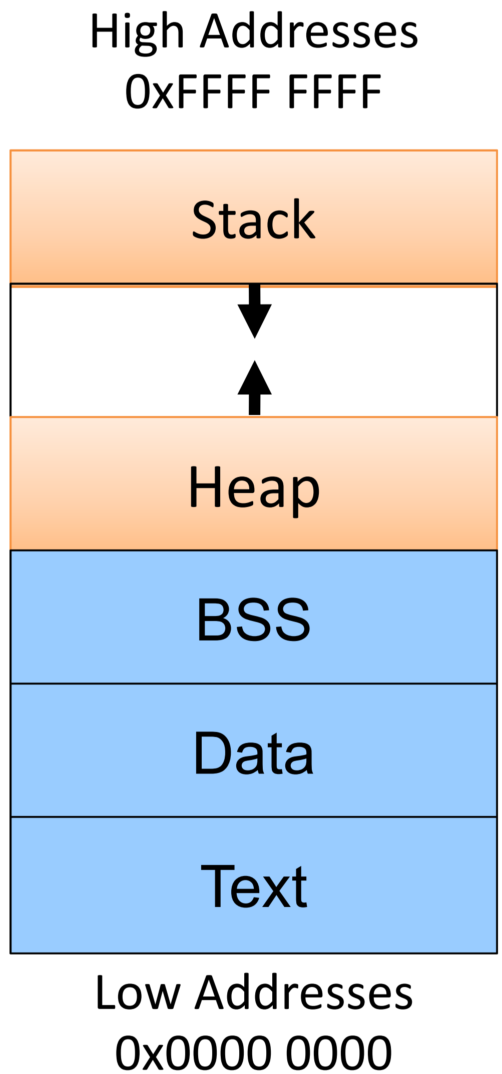
# Virtual Memory




Mapping virtual addresses to real addresses

# Unix Address Space

- Text: machine code of the program, compiled from the source code
- Data: static program variables initialized in the source code prior to execution
- BSS (block started by symbol): static variables that are uninitialized
- Heap : data dynamically generated during the execution of a process
- Stack: structure that grows downwards and keeps track of the activated method calls, their arguments and local variables



# Vulnerabilities and Attack Method

- Vulnerability scenarios
  – The program has root privileges (setuid) and is launched from a shell
  – The program is part of a web application
- Typical attack method
  1. Find vulnerability
  2. Reverse engineer the program
  3. Build the exploit

# Buffer Overflow Attack in a Nutshell

- First described in

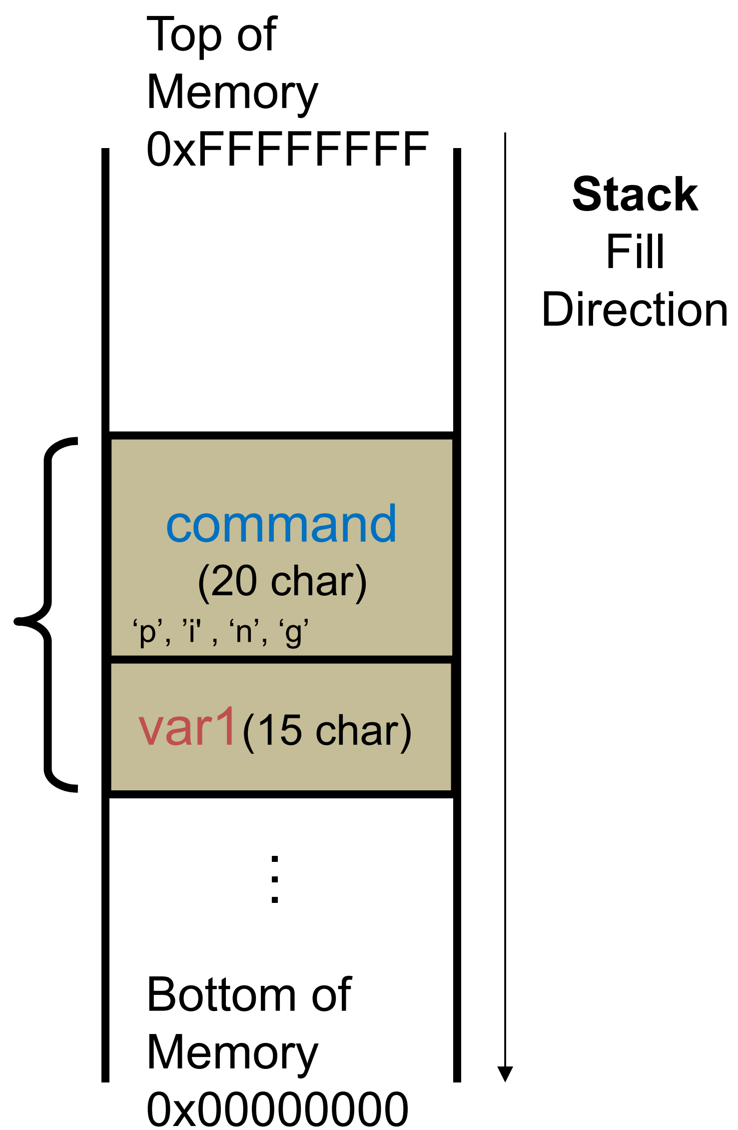
  Aleph One. Smashing The Stack For Fun And Profit. e-zine www.Phrack.org #49, 1996

- The attacker exploits an unchecked buffer to perform a buffer overflow attack
- The ultimate goal for the attacker is getting a shell that allows to execute arbitrary commands with high privileges
- Kinds of buffer overflow attacks:
  - Heap smashing
  - Stack smashing

# Buffer Overflow

**myping.c**

```
main(intargc, char *argv[ ])
/* get user_input */
{
   char command[20];
   char var1[15];
strcpy(command, “ping ");
strcat(command, argv[1]);
strcpy(var1, argv[1]);
printf(“The result of myping %s is:\n”,
var1);
system(command);
}
```

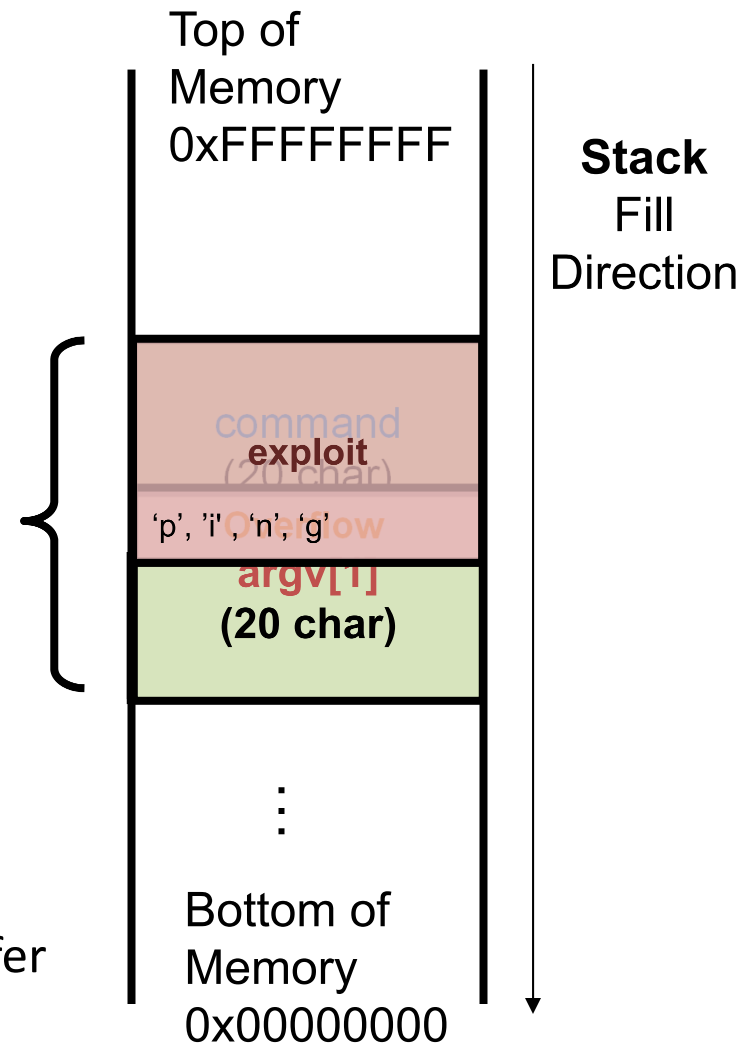



- A simple wrapper around ping

# strcpy() Vulnerability

**mying.c**

```
main(intargc, char *argv[ ])
/* get user_input */
{
   char command[20];
   char var1[15];
strcpy(command, “ping ");
strcat(command, argv[1]);
strcpy(var1, argv[1]);
printf(“The result of myping %s is:\n”,
var1);
system(command);
}
```



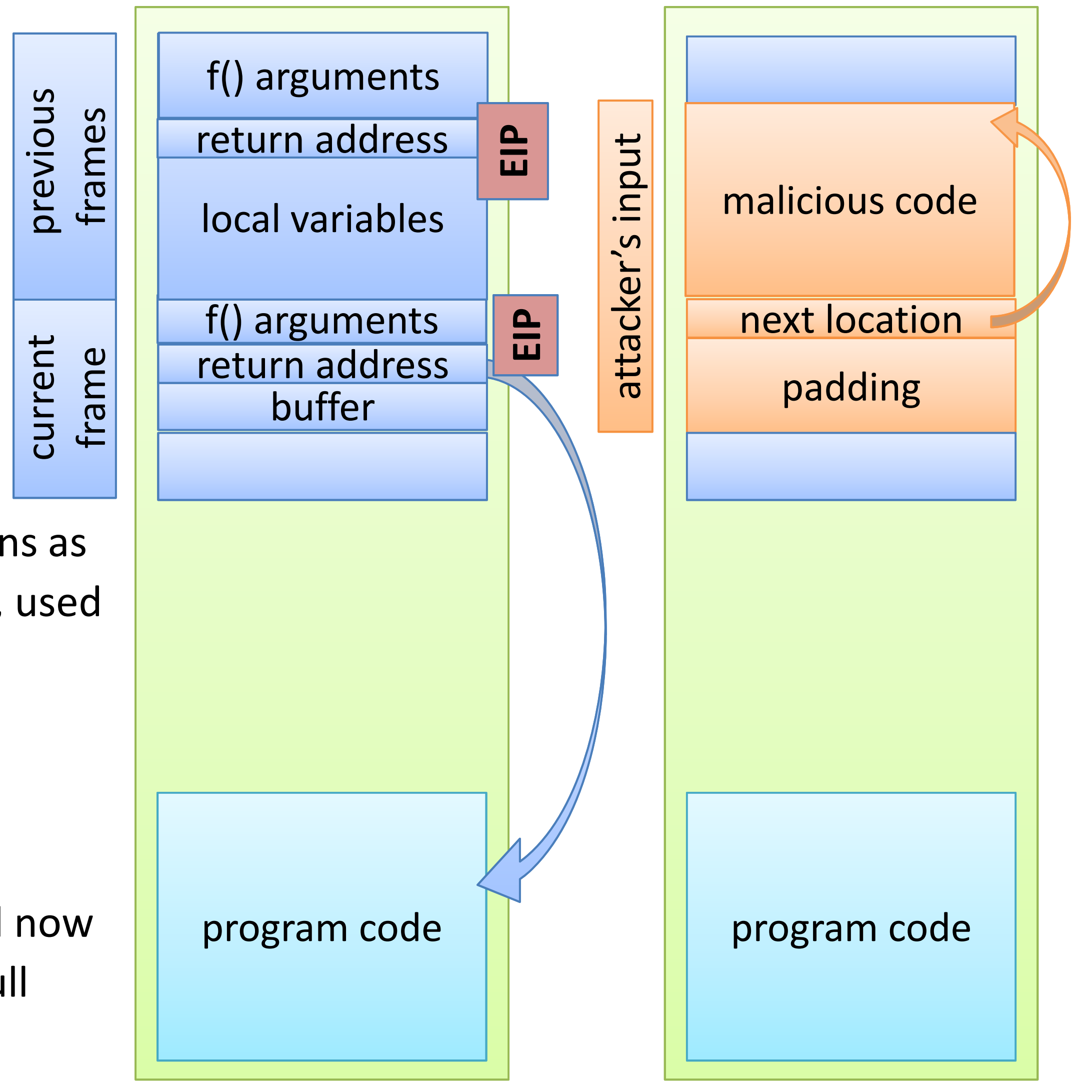

- argv[1] is the user input
- strcpy(dest, src) does not check dest buffer
- strcat(d, s) concatenates strings

# strcpy() vs. strncpy()

- Function strcpy()copies the string in the second argument into the first argument
  - e.g., strcpy(dest, src)
  - If source string > destination string, the overflow characters may occupy the memory space used by other variables
  - The null character is appended at the end automatically
- Function strncpy() copies the string by specifying the number **n** of characters to copy
  - e.g., strncpy(dest, src, n);dest[n] = ‘\0’
  - If source string is longer than the destination string, the overflow characters are discarded automatically
  - You have to place the null character manually

# Return Address Smashing

```
void fingerd (…) {
        char buf[80];
        …
        get(buf);
        …
}
```




- The Unix fingerd() system call, which runs as root (it needs to access sensitive files), used to be vulnerable to buffer overflow
- Write malicious code into buffer and overwrite return address to point to the malicious code
- When return address is reached, it will now execute the malicious code with the full rights and privileges of root

# Unix Shell Command Substitution

- The Unix shell enables a command argument to be obtained from the standard output of another
- This feature is called command substitution
- When parsing command line, the shell replaces the output of a command between back quotes with the output of the command
- Example:
  - File name.txt contains string farasi
  - The following two commands are equivalent
  - finger \`cat name.txt\`
  - finger farasi

# Shellcode Injection

- An exploit takes control of attacked computer so injects code to “spawn a shell” or “shellcode”
- A shellcode is:
  - Code assembled in the CPU’s native instruction set (e.g. x86 , x86-64, arm, sparc, risc, etc.)
  - Injected as a part of the buffer that is overflowed.
- We inject the code directly into the buffer that we send for the attack
- A buffer containing shellcode is a “payload”

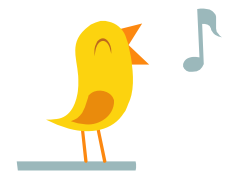
# Buffer Overflow Mitigation

- We know how a buffer overflow happens, but why does it happen?
- This problem could not occur in Java; it is a C problem
  - In Java, objects are allocated dynamically on the heap (except ints, etc.)
  - Also cannot do pointer arithmetic in Java
  - In C, however, you can declare things directly on the stack
- One solution is to make the buffer dynamically allocated
- Another (OS) problem is that fingerd had to run as root
  - Just get rid offingerd's need for root access (solution eventually used)
  - The program needed access to a file that had sensitive information in it
  - A new world-readable file was created with the information required by fingerd

# Stack-based buffer overflow detection using a random canary

Normal (safe) stack configuration:

| | | | | |
|---|---|---|---|---|
| Buffer | Other local variables | Canary (random) | Return address | Other data |



Buffer overflow attack attempt:

| | | | |
|---|---|---|---|
| Buffer | Overflow data | Corrupt return address | Attack code |


- The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.

# Cross Site Scripting (XSS)

- Attacker injects scripting code into pages generated by a web application
  - Script could be malicious code
  - JavaScript (AJAX!), VBScript, ActiveX, HTML, or Flash
- Threats:
  - Phishing, hijacking, changing of user settings, cookie theft/poisoning, false advertising , execution of code on the client, ...

# XSS Example

- Website allows posting of comments in a guestbook
- Server incorporates comments into page returned

```
<html>
<body>
<title>My Guestbook!</title>
Thanks for signing my guestbook!<br />
Here's what everyone else had to say:<br />
Joe: Hi! <br />
John: Hello, how are you? <br />
Jane: How does this guestbook work? <br />
</body>
```

- Attacker can post comment that includes malicious JavaScript

```
Evilguy: <script>alert("XSS Injection!");
    </script> <br />
```

guestbook.html

```
<html>
<title>Sign My Guestbook!</title>
<body>
Sign my guestbook!
<form action="sign.php"
    method="POST">
<input type="text" name="name">
<input type="text" name="message"
    size="40">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

# SQL Injection

- SQL injection occurs when user input is not filtered for escape characters and is then passed into an SQL statement
- uName = getRequestString("username");
  uPass = getRequestString("userpassword");

  sql = 'SELECT * FROM Users WHERE Name =" ' + uName + ' " AND Pass =" ' + uPass + ' " '
- Normal case:
- SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
- What if the user inputs User Name as " or ""=" and Password as " or ""="
- We get
- SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""

| The condition is satisfied because **or ""=""** is always TRUE |
|---|

# Definitions

- *Vulnerability (security flaw)*: failure of security policies, procedures, and controls that allow a subject to commit an action that violates the security policy
  - Subject is called an *attacker*
  - Using the failure to violate the policy is *exploiting the vulnerability* or *breaking in*

# Incomplete List of Vulnerabilities – Latest Version

- Buffer overflows/memory safety errors
- Integer overflow, underflow, and sign conversion errors
- Null pointer errors
- Race conditions, atomicity violations, TOCTTOU (time of check to time of use) vulnerabilities
- Resource drains that may be subject to attack (e.g., memory or file handle leaks)
- Other memory errors (double frees, use after free, uninitialized memory use)
- Insufficient access control checks/incomplete mediation
- Tainted data/input validation errors (e.g., format string vulnerabilities, lack of checking on command strings sent to OS, insufficient restrictions on input file names)
- Unhandled exceptions or returned error status codes
- Leaks of confidential information

# Classification Scheme by the Research Into Secure Operating Systems (RISOS) Project

- Incomplete parameter validation
- Inconsistent parameter validation
- Implicit sharing of privileged/confidential data
- Asynchronous validation/inadequate serialization
- Inadequate identification/authentication/authorization
- Violable prohibition/limit
- Exploitable logic error

# Incomplete Parameter Validation

- Parameter not checked before use
- Example: emulating integer division in SunOS kernel (RISC chip involved)
  - Caller provided addresses for quotient, remainder
  - Quotient address checked to be sure it was in user's protection domain
  - Remainder address *not* checked
    - Set remainder address to address of process' level of privilege
    - Compute 25/5 and you have level 0 (kernel) privileges
- Solution: check for type, format, range of values, access rights, presence (or absence)

# Inconsistent Parameter Validation

- Each routine checks parameter is in proper format for that routine but the routines require different formats
- The inconsistency across interfaces causes this flaw

- Example: each database record is one line, with colons separating fields
  - One program accepts colons, newlines as part of data within fields
  - Another program reads them as field and record separators
  - This allows bogus records to be entered

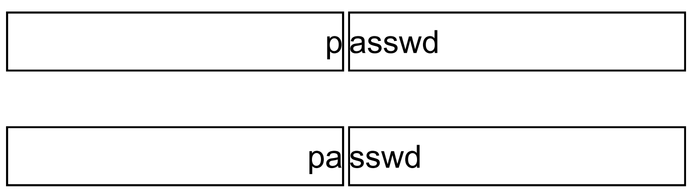
# Implicit Sharing of Privileged / Confidential Data

- OS does not isolate users / processes properly
- Example: attacking file password protection in TENEX
  - OS allows user to determine when paging occurs
  - Files protected by passwords
    - Passwords checked character by character; stops at first incorrect one
  - Attack:

    (1) position guess for password so page boundary occurred between 1st and 2nd character

    - If no page fault, 1st char was wrong; if page fault, it was right

    (2) Continue until password discovered



# Asynchronous Validation / Inadequate Serialization

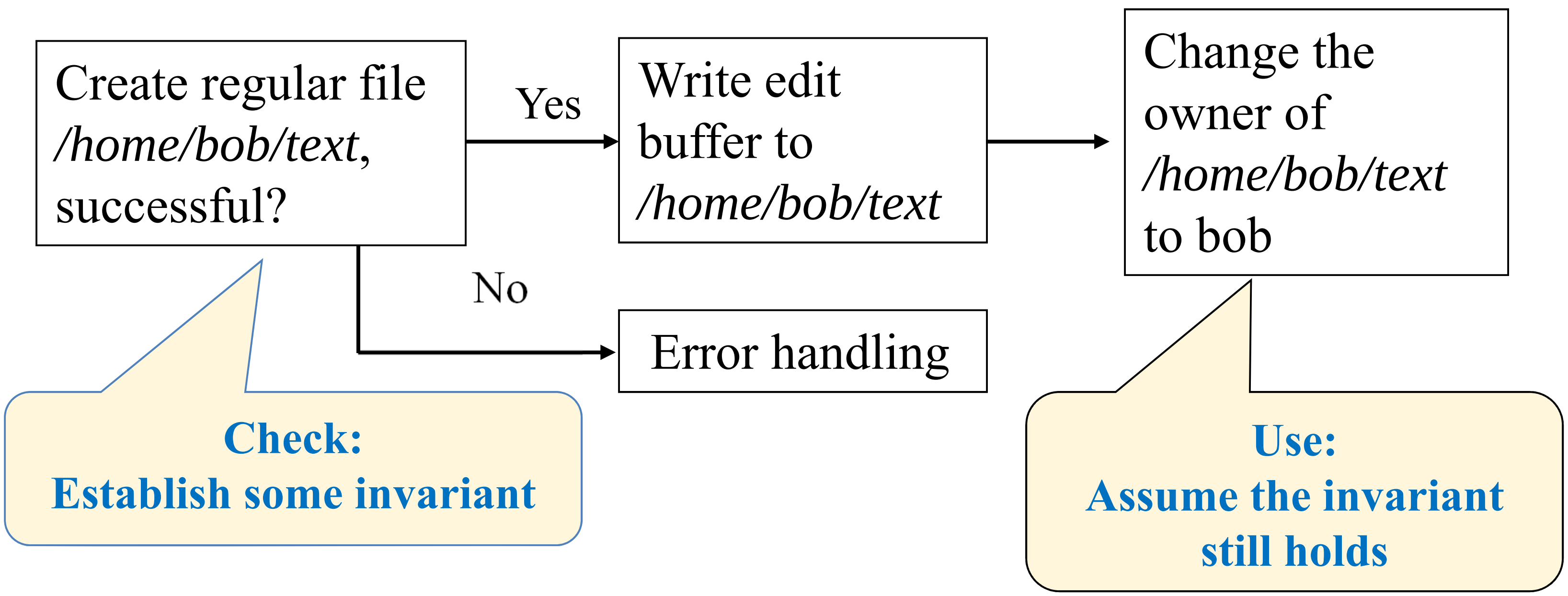
- Time of check to time of use (TOCTTOU) flaws, intermixing reads and writes to create inconsistencies
- Example: *vi* flaw discussed next

# Definition of TOCTTOU

- Time-Of-Check-To-Time-Of-Use, a race condition in Unix-style file systems.
- Check - establishes some precondition (invariant) about a file.
- Use - Operates on the file assuming that the invariant is still valid.

# TOCTTOU Call Pair: vi

- Runs as root to edit a file owned by a normal user. E.g., vi /home/bob/text, then saves the file.



# TOCTTOU Attack Scenario: vi

Effect: The attacker may get unauthorized root access!

vi (root)

Create regular file */home/bob/text*, successful?

Yes

Write …

Change the owner of */home/bob/text* (actually /etc/passwd) to bob

Time

bob

Delete */home/bob/text*

Create symbolic link */home/bob/text*, which points to /etc/passwd

# Inadequate Identification / Authorization / Authentication

- Erroneously identifying user, assuming another's privilege, or tricking someone into executing program without authorization

- Example: on UNIVAC 1100, access to a file named "SYS$*DLOC$" meant process privileged
  - Check: can process access any file with qualifier name beginning with "SYS" and file name beginning with "DLO"?
  - There was an *ordinary* file named "SYSA*DLOC$", which can be accessed by any process. Therefore, an attacker just needs to access this file and her process is privileged

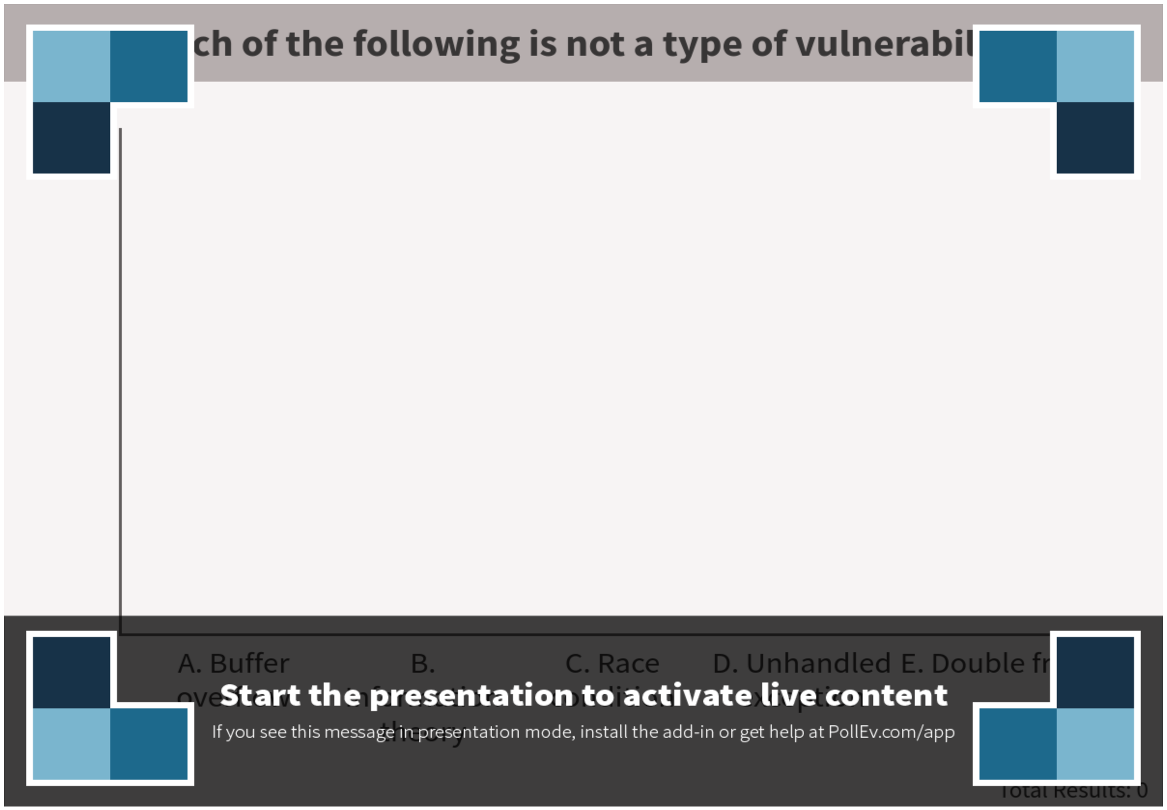
# Violable Prohibition / Limit

- Boundary conditions not handled properly
- Example: early versions of the TENEX OS kept in low memory, user process in high memory
  - Boundary was highest address of OS: os_top
  - All memory accesses by user apps checked against this:

    if addr > os_top, okay
  - Memory addresses beyond end of high memory are reduced modulo memory size
  - So, process could access (memory size)+1, which is greater than os_top and thus allowed
  - But actually the hardware accesses word 1, which is part of OS …

# Exploitable Logic Error

- Problems not falling into other classes
  - Incorrect error handling, unexpected side effects, incorrect resource allocation, etc.
- Example: unchecked return from monitor
  - Monitor adds 1 to address in user's return word, returns to that address
    - Index bit (indicating indirection) is a bit in word
    - Attack: set return word to be –1; adding 1 overflows, changes index bit, so return is to location stored in register 1
  - Arrange for this to point to bootstrap program (of the attacker) stored in other registers
    - On return, program executes with system privileges


ch of the following is not a type of vulnerabil
A. Buffer
B.
C. Race
D. Unhandled
E. Double fr
Start the presentation to activate live content
If you see this message in presentation mode, install the add-in or get help at PollEv.com/app
Total Results: 0

# How to Detect Vulnerabilities?

- Formal verification
- Penetration testing

# Formal Verification

- Mathematically verifying that a system satisfies certain constraints

- *Preconditions* state assumptions about the system

- *Postconditions* are result of applying system operations to preconditions, inputs

- Required: postconditions satisfy constraints

# Penetration Testing

- Testing to verify that a system satisfies certain constraints
- Hypothesis stating system characteristics, environment, and state relevant to vulnerability
- Result is compromised system state
- Apply tests to try to move system from state in hypothesis to compromised system state (constraints violated)

# Penetration Studies

- Test for evaluating the strengths and effectiveness of all security controls on system
  - Also called *tiger team attack* or *red team attack*
  - Goal: violate site security policy
  - Not a replacement for careful design, implementation, and structured testing
  - Tests system *in toto*, once it is in place
    - Includes procedural, operational controls as well as technological ones

# Flaw Hypothesis Methodology

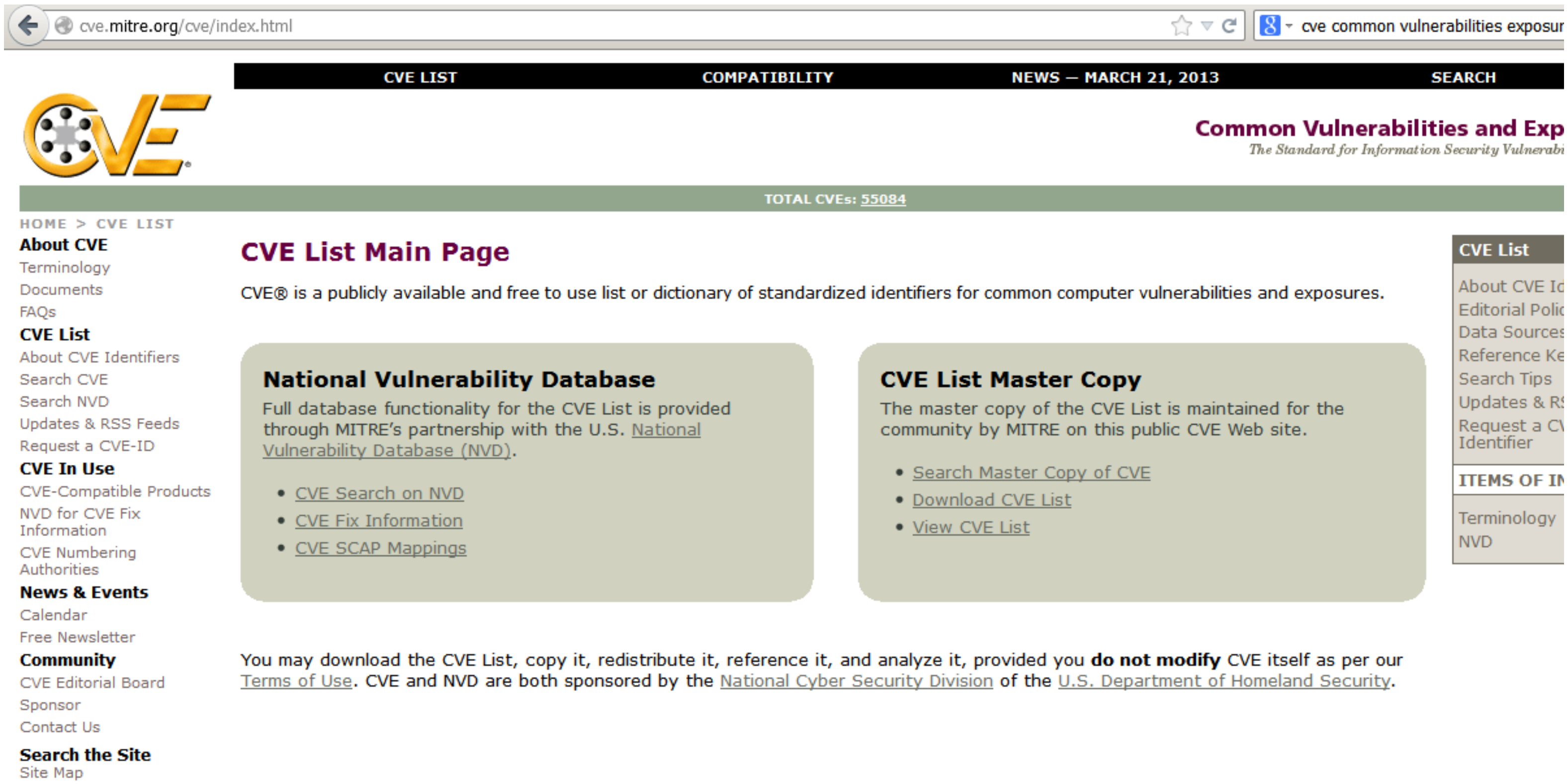
1. Information gathering
   - Become familiar with system's functioning
2. Flaw hypothesis
   - Draw on knowledge to hypothesize vulnerabilities
3. Flaw testing
   - Test them out
4. Flaw generalization
   - Generalize vulnerability to find others like it
5. (*maybe*) Flaw elimination
   - Testers eliminate the flaw (usually *not* included)

# Metasploit Framework

- Network Scanning and Host Fingerprinting:
  - Find active hosts on the network (IP address)
  - Find vulnerable hosts (OS, services, open ports, etc)
  - Nmap
- Testing whether a target host has a known vulnerability
  - E.g., CVE-2008-4250

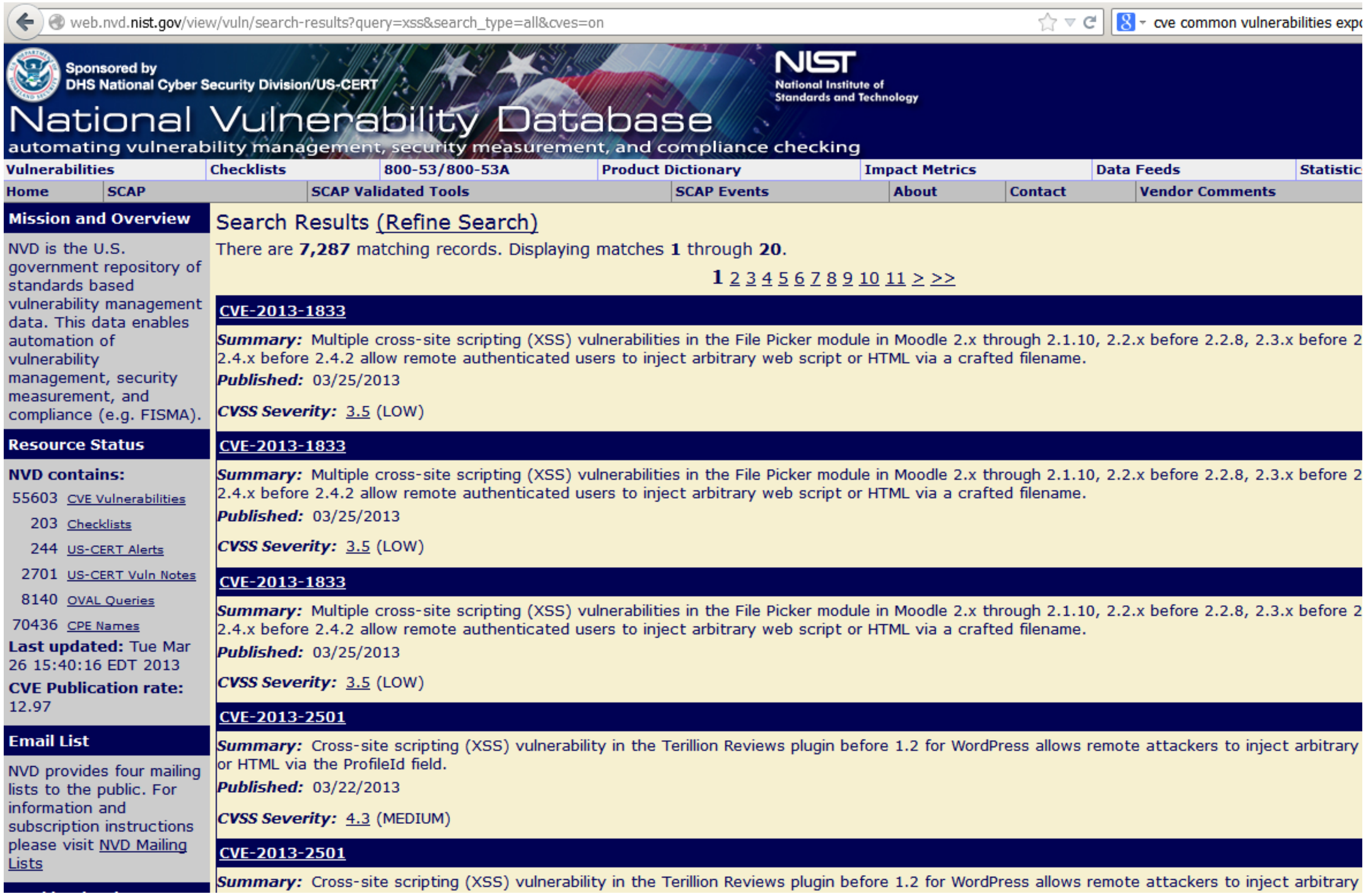
# Common Vulnerabilities and Exposures

- CVE (http://cve.mitre.org/) is a dictionary of publicly known information security vulnerabilities and exposures

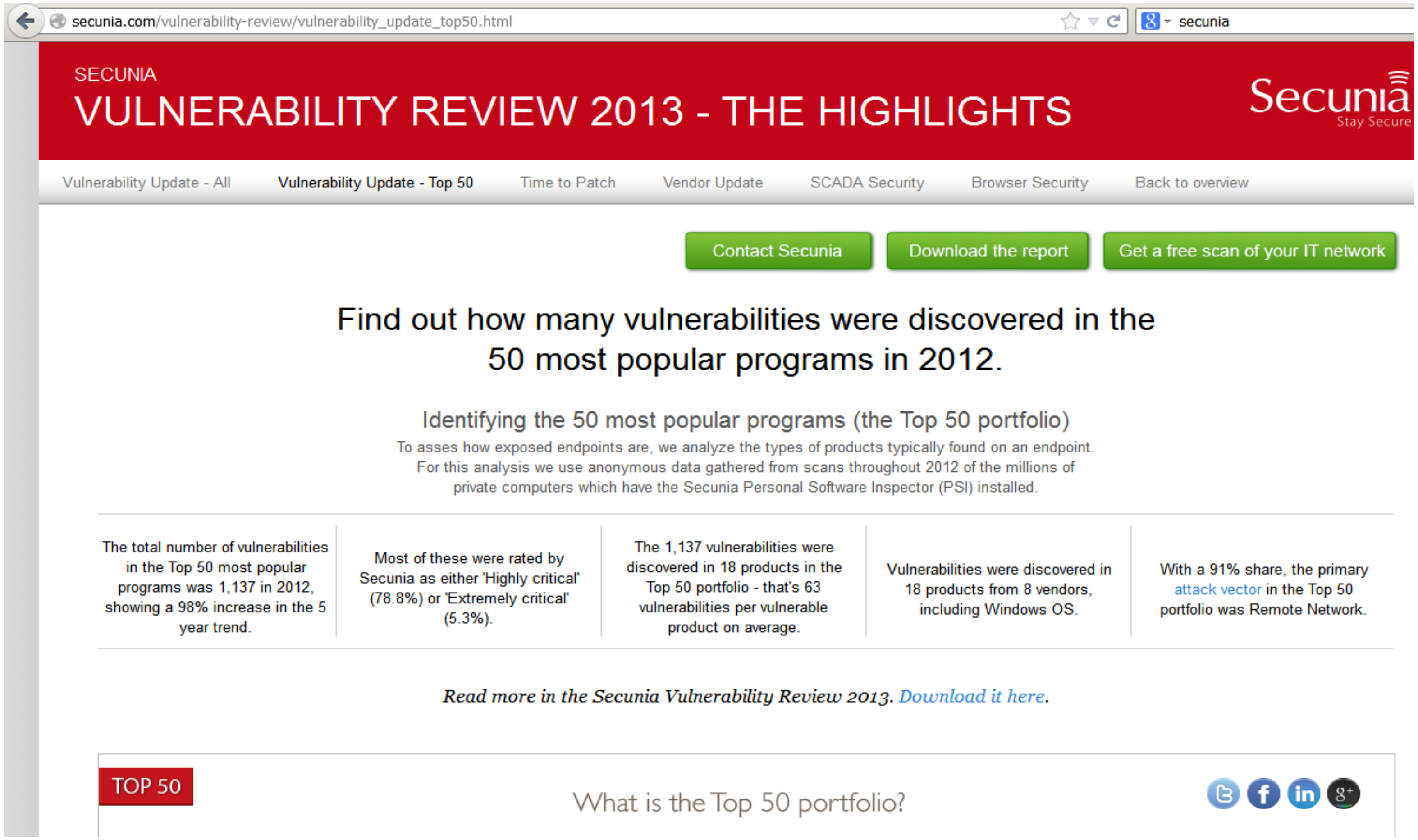


# NVD (National Vulnerability Database)

## http://nvd.nist.gov/



# Secunia




- https://secuniaresearch.flexerasoftware.com/community/research/