Elements of Security

Program Security and Viruses

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: April 28, 2015

Classifying Bugs

The IEEE (IEEE standard 729) has suggested a standard terminology for “bugs” in computer programs.

Fault:	a fault is a defect that gives rise to an error. It could be due to defective, missing or extra instructions, or a manufacturing flaw in hardware.
Error:	a detectable deviation from the agreed specification or requirements. An error is caused by a *fault* and may lead to a failure.
Failure:	the delivered service deviates from the specified service, where the service specification is an agreed description.

Parhami's Taxonomy

Behrooz Parhami gives the following expanded taxonomy of system states:

Ideal:	the program perfectly meets its specification.
Defective:	there is some flaw in the hardware or software.
Faulty:	certain system states may expose the defect, resulting in incorrect values or decisions.
Erroneous:	the fault is actually exercised, leading to an error.
Malfunction:	an error may cause observed deviation from the specification.
Degraded:	perceivable service-level effoect
Failure:	catastrophic or unsafe system behavior, or termination of system action.

Fixing Faults

Number of faults detected and fixed is not a reliable measure of software quality. Hence, the paradigm of *penetrate and patch* is not a good way to build secure systems.

Patch efforts often make a system *less* secure than before, because the patches introduce new faults.

- Effort focuses narrowly on the fault without correcting the underlying design or requirements flaws.
- A fault may have nonobvious side effects far removed from the location of the fault.
- A fault may not be fixed properly because it would impact system functionality or performance.

Security Flaws

Program security flaws can arise from many different types of faults, including intentionally malicious code, and code developed in sloppy or misguided ways. We often divide program flaws into these two categories.

- Intentional attacks (called **cyber attacks**) get more press, but
- inadvertant errors undoubtedly cause much more damage.

Eliminating Security Flaws

It is probably impossible to completely eliminate security flaws.

1. Program controls operate at the level of individual programs and programmers. Security is a system-wide phenomenon that results from the complex interaction of many parts of the system.
2. Software engineering evolves more quickly than does computer security. That means that security is always trying to catch up with the state of the art in software design.

OWASP Top 10 Web Vulnerabilities

The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. They regularly publish a 10 top list of security vulnerabilities *for web applications*. Below is a recent list:

Cross Site Scripting (XSS): XSS flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content.

Injection Flaws: Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data.

OWASP Top 10 Web Vulnerabilities

Malicious File Execution: Code vulnerable to remote file inclusion (RFI) allows attackers to include hostile code and data, resulting in devastating attacks, such as total server compromise.

Insecure Direct Object Reference: A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter.

Cross Site Request Forgery (CSRF): A CSRF attack forces a logged-on victim's browser to send a pre-authenticated request to a vulnerable web application, which then forces the victim's browser to perform a hostile action to the benefit of the attacker.

OWASP Top 10 Web Vulnerabilities

Information Leakage and Improper Error Handling: Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Attackers use this weakness to steal sensitive data, or conduct more serious attacks.

Broken Authentication and Session Management: Account credentials and session tokens are often not properly protected. Attackers compromise passwords, keys, or authentication tokens to assume other users' identities.

Insecure Cryptographic Storage: Web applications rarely use cryptographic functions properly to protect data and credentials. Attackers use weakly protected data to conduct identity theft and other crimes, such as credit card fraud.

OWASP Top 10 Web Vulnerabilities

Insecure Communications: Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications.

Failure to Restrict URL Access: Frequently, an application only protects sensitive functionality by preventing the display of links or URLs to unauthorized users.

Nonmalicious Program Errors

Landwehr et al. give a taxonomy of program flaws that might result in security lapses. The inadvertant flaws fall into the following categories:

- validation error (incomplete or inconsistent)
- domain error
- serialization or aliasing
- inadequate identification or authentication
- boundary condition violation
- other exploitable logic errors

Validation: Buffer Overflow

A buffer is a bounded memory space in which data is held. Writing beyond the end of a buffer may:

- be detected by the compiler or run-time system;
- may effect adjacent user data space;
- may effect adjacent user program space;
- may effect adjacent system data space;
- may effect adjacent system program space.

An overflow into system space may allow an attacker to insert system code running at system permission level, modify the call stack, etc. Both the Internet worm (1988) and the Code Red virus (2001) used buffer overflow in critical ways.

Smashing the Stack

Buffer overflows account for over 50% of advisories published by CERT (computer security incident report team):

Morris worm (1988): overflow in fingerd, infected 10% of the existing Internet.

Code Red (2001): overflow in MS-IIS server, 300,000 machines infected in 14 hours.

SQL Slammer (2003): overflow in MS-SQL server, 75,000 machines infected in 10 minutes.

Attacks on Buffers

A **buffer** is a data storage area inside memory (stack or heap).

- Buffers are intended to hold a pre-defined amount of data. If more is stuffed into it, it may spill into adjacent memory.
- If executable code is supplied as “data,” the machine may be fooled into executing it.

An attack can exploit *any* memory operation: pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines.

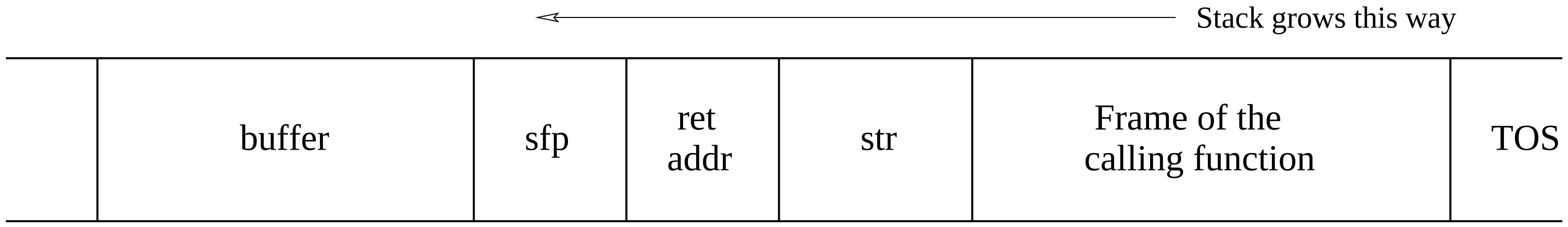
Why are buffer overflows more prevalent in C than in Java? Is the solution to just use Java instead of C?

Stack Buffers

Consider the following function:

```
void func (char *str) {
  char buf[126];
  strcpy (buf, str);
}
```

When this function is invoked, a new frame is pushed onto the stack.

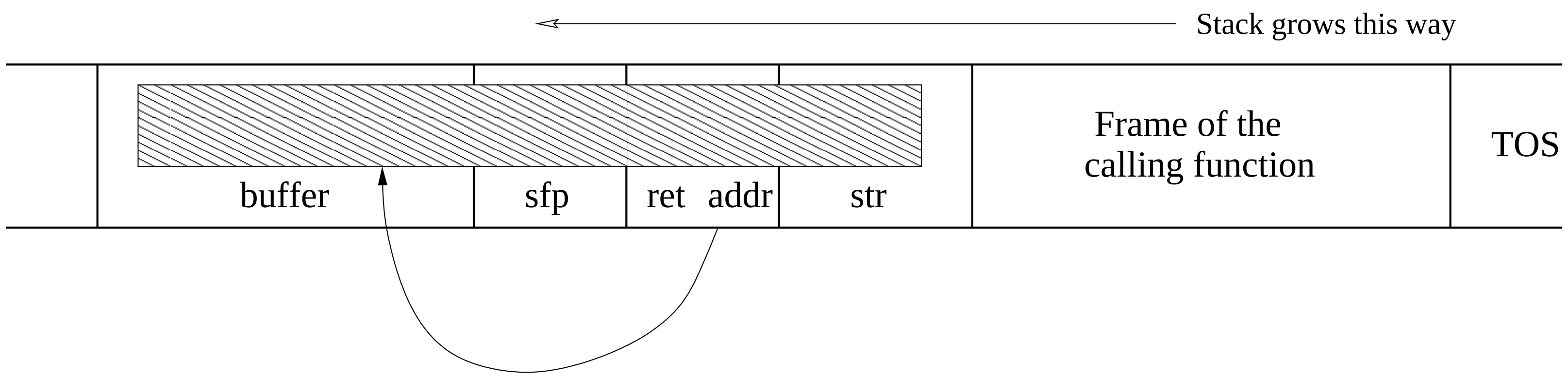


Stack Buffers

If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations, including the frame pointer and return address.

If buffer value contains attacker-generated code and ret addr points into this code, the attacker can cause this to be executed.

If the running program has root privileges, so will the new code. The attacker can, e.g., spawn a new shell with root privilege.



Overflow Issues

- Why not just make the stack non-executable? There are some circumstances where we want to treat data as code.
- Overflow portion of the buffer must contain the correct address of the attack code in the RET position. This is harder than it seems.
- Many C functions do not check input size: strcpy, strcat, gets, scanf, printf, ...

Overflow Attacks

In addition to buffers on the stack, there are related attacks:

- overflow of buffers allocated on the *heap* may change pointers to important data or cause a crash;
- *function pointer overflow* may cause an alternative function to be executed;
- *integer overflow* of a variable used in a bounds computation may facilitate an attack.

Incomplete Mediation

Suppose you fill out a form on a web page. The result may be packaged and sent to the server side as, for example:

```
http://www.somesite.com/subpage/userinput
&parm1=(808)555-1212&parm2=2004Jan01
```

What happens if you insert nonsense for the parameters clearly intended to contain a phone number and date?

If the values are checked on the *client side* (i.e., by code in the browser), the data fields may still be tampered with before the line is sent. The data is not completely *mediated*.

Time-of-Check / Time-of-Use

If a security access check is performed significantly before the access is actually performed, an attacker may perform a "bait and switch." This is also called a *serialization* or *synchronization* flaw.

Suppose a user presents a request in the form of a pair: *[file-name, access]*. The system checks whether the access is allowed. If the pair is left in user space, the user might alter the file name while the access check is occurring, and obtain access to a file for which he does not have appropriate permissions.

This is a particular concern in *capability-based* systems, in which users maintain "tickets" granting them access rights. These tickets must be unalterable and unforgeable.

Malicious Code

The Computer Emergency Response Team (CERT) at Carnegie Mellon University, tracks vulnerabilities and attacks.

Period	**Vulnerabilities**	**Incidents**
1998	262	3,734
1999	417	9,859
2000	1,090	21,756
2001	2,437	52,658
2002	4,129	82,095
2003	3,784	137,529
2004	3,780	**
2005	5,990	**
2006	8,064	**

"** Given the widespread use of automated attack tools, attacks ... have become so commonplace that counts of the number of incidents provide little information with regard to assessing the scope and impact of attacks."

Malicious Code (Cont.)

Malicious code (viruses, worms, etc.) runs with the permissions of the user or operating system, and can do anything that a legitimate user can do–create files, write to files, delete data and files, etc.

It may also sit dormant until triggered by some event, including reaching a certain time.

Malicious Code Taxonomy

Malicious code or a **rogue program** is the generic name for unanticipated or undesired effects in programs, caused by an agent intent on damage. The **agent** is the author or distributor of the program.

Virus: a program that can pass on malicious code to other nonmalicious programs by modifying them.

- A **transient** virus runs when its attached program executes and terminates when the attached program ends.
- A **resident** virus locates itself in memory and can run as a stand-alone program.

Trojan horse: malicious code that, in addition to its primary effect, has a second, nonobvious malicious effect.

Malicious Code Taxonomy

Logic bomb:	class of code that “detonates” on a specific trigger.
Time bomb:	logic bomb whose trigger is a time or date.
Trapdoor or backdoor:	program to allow access other than by the obvious, direct call, perhaps with special privileges.
Worm:	program that spreads copies of itself through a network.
Rabbit:	virus or worm that replicates without bound, with the intention of exhausting system resources.

How Viruses Attach

For a virus to operate, it must be executed. There are many ways to ensure that virus code will be executed.

- The virus may be resident within the code of another program. When execution reaches that point in the code, the virus code is executed.
- The virus can be embedded in an executable attachment to an email message.
- The execution might be triggered by a specific time or event.

How Viruses Attach

- The virus may be appended at the beginning of a program. When the program is invoked, the virus runs first, and then may transfer control to the original program.
- The virus may surround a program, i.e., have portions both before and after the program to ensure that it regains control after the program runs.
- The virus may be integrated into the program, altering the functionality of the program. This requires intimate knowledge of the program structure.

Qualities of Viruses

A virus writer may strive for some subset of the following characteristics:

- It is hard to detect.
- It is not easily destroyed or deactivated.
- It spreads infection widely.
- It can reinfect its home program or other programs.
- It is easy to create.
- It is (relatively) machine independent and OS independent.

Most viruses execute only once, and do their damage during this execution.

Boot Sector Viruses

The bootstrap loader is a small piece of code that runs when your machine is rebooted. The goal is to load the operating system from disk, often by "chaining" together blocks. Each block loaded contains the location of the subsequent block.

A **boot sector virus** interrupts the chain and causes loading of virus code rather than regular OS code. This has the following effects:

- The virus seizes control of the OS very early and has complete control of the machine.
- Since OS files are often made invisible to ordinary users, the take-over may go unnoticed.

Memory-Resident Viruses

Most programs are swapped into memory to run. After they run, the space is re-used for other programs. Some OS routines run so frequently that they are kept in memory. These are called TSR's or "terminate and stay resident" routines.

A virus that infects a TSR is guaranteed to be activated many times. This is useful if the purpose of the virus is to infect media that may be mounted and removed. Eg., each time the virus runs, it can check whether any disk, floppy, CD, etc. has been mounted and, if so, infect that medium.

Other Homes for Viruses

A virus may infect:

- *Application Programs* such as word processors and spreadsheets. These often have startup macros executed each time the application is invoked.
- *Libraries* may be used by many other programs, and shared and transmitted by many users.
- *Other applications* such as compilers, loaders, linkers, runtime monitors, debuggers and even virus control programs, may be infected and shared widely.

Virus Effects and Causes

Virus Effect	How it is Caused
Attach to executable program	Modify file directory Write to executable program file
Attach to disk or control file	Modify directory Rewrite data Append to data Append data to self
Remain in memory	Intercept interrupt Load self into nontransient memory
Infect disks	Intercept interrupt Intercept OS call (format disk, eg.) Modify system file Modify ordinary executable program

Virus Effects and Causes

Virus Effect	**How it is Caused**
Conceal self	Intercept system calls and falsify result
	Classify self as “hidden” file
Spread infection	Infect boot sector
	Infect systems programs
	Infect ordinary programs
	Infect data programs use to control execution
Prevent deactivation	Activate before deactivating program
	Store copy to reinfect after deactivation

Detecting Viruses

Detecting viruses is undecidable, in general. Nevertheless, virus scanners look for *signatures*, certain recognizable patterns.

Common text strings: once a virus has been identified, it may be recognized by a characteristic string in the code.

Storage patterns: a virus that attaches to a file may cause the file size to grow in a predictable way, or may invalidate the checksum of the file.

Execution patterns: nonstandard patterns of creation or deletion of files may signal the presence of a virus.

Detecting Viruses

To avoid detection some virus writers use multiple forms of the virus, so that the virus scanner may have to look for a different signature for each form. Such viruses are called **polymorphic**.

Some ways to fool scanners include:

- Reorder the virus code, using JUMPs between blocks.
- Intersperse harmless instructions randomly throughout the code.
- *Encrypt* copies of the virus using different keys. In this case, the call to the decryption library routine must be in the clear and can serve as a signature.

Preventing Infection

The following are some techniques for avoiding infection:

- Use only commercial software acquired from reliable vendors.
- Test all new software on an isolated computer.
- Open attachments only when you know them to be safe.
- Make a recoverable system image and store it safely.
- Make and retain backup copies of executable system files.
- Use virus detectors regularly and update them daily.