Software Security

Secure Software

Software is secure if it can handle intentionally malformed input; the attacker picks (the probability distribution of) the inputs.

Secure software: protect the integrity of the runtime system.

Secure software ≠ software with security features.

Networking software is a popular target:

- intended to receive external input.
- involves low level manipulations of buffers.

Security & Reliability

Reliability deals with accidental failures: failures are assumed to occur according to some given probability distribution.

The probabilities for failures is given first, then protection mechanisms are constructed.

To make software more reliable, it is tested against typical usage patterns: “It does not matter how many bugs there are, it matters how often they are triggered”.

Security & Reliability

In security, the defender has to move first; the attacker then picks inputs that exploit weak defences.

To make software more secure, it has to be tested against "untypical" usage patterns (but there are typical attack patterns).

On a stand-alone PC, you are in control of the software components sending inputs to each other.

On the Internet, hostile parties provide input:

Do not "trust" your inputs.

Agenda

Malware

Dangers of abstraction

Input validation

Integers

Buffer overflows

Scripting languages

Race conditions

Defences: Prevention – Detection – Reaction

Malware

Malware: software that has a malicious purpose.

Computer virus: self-replicating code attached to some other piece of code; a virus infects a program by inserting itself into the program code.

Worm: replicating but not infecting program; most reported virus attacks would be better described as worm attacks.

Trojan horse: program with hidden side effects, not intended by the user executing the program.

Logic bomb: program that is only executed when a specific trigger condition is met.

Preliminaries

When writing code, programmers use elementary concepts like character, variable, array, integer, data & program, address (resource locator), atomic transaction, …

These concepts have abstract meanings.

For example, integers are an infinite set with operations ‘add’, ‘multiply’, ‘less or equal’, …

To execute a program, we need concrete implementations of these concepts.

Benefits of Abstraction

Abstraction (hiding ‘unnecessary’ detail) is an extremely valuable method for understanding complex systems.

We don’t have to know the inner details of a computer to be able to use it.

We can write software using high level languages and graphical methods.

Anthropomorphic images explain what computers do (send mail, sign document).

Dangers of Abstraction

Software security problems typically arise when concrete implementation and the abstract intuition diverge.

We will explore a few examples:

- Address (location)
- Character
- Integer
- Variable (buffer overflows)
- Atomic transaction

Input Validation

An application wants to give users access only to files in directory **`A/B/C/`**.

Users enter filename as **`input`**; full file name constructed as **`A/B/C/input`**.

Attack: use `../` a few times to step up to root directory first; e.g. get password file with input **`/../../../../etc/passwd`**.

Countermeasure: input validation, filter out `../` (but as you will see in a moment, life is not that easy).

Do not trust your inputs.

Unicode Characters

UTF-8 encoding of Unicode characters [RFC 2279]

Multi-byte UTF-8 formats: a character has more than one representation

Example: “/”

	format	binary	hex
1 byte	0xxx xxxx	0010 1111	2F
2 byte	110x xxxx	1100 0000	C0
	10xx xxxx	1010 1111	AF
3 byte	1110 xxxx	1110 0000	E0
	10xx xxxx	1000 0000	80
	10xx xxxx	1010 1111	AF

Exploit “Unicode bug”

Vulnerability in Microsoft IIS; URL starting with {IPaddress}/scripts/..%c0%af../winnt/system32/

Translated to directory C:\winnt\system32

- The /scripts/ directory is usually C:\inetpub\scripts
- Because %c0%af is the 2 byte UTF-8 encoding of /
- ..%c0%af../ becomes ../../
- ../../ steps up two levels in the directory

IIS did not filter illegal Unicode representations that use multi-byte UTF-8 formats for single byte characters.

Double Decode

Consider URL starting with {addr.}/scripts/..%25%32%66../winnt/system32/

This URL is decoded to {addr.}/scripts/..%2f../winnt/system32/

- Convert %25%32%66 to Unicode:
  00100101 00110010 01100110 → %2f (= /)

If the URL is decoded a second time, it gets translated to directory C:\winnt\system32

Characters change their meaning “by the act of observation”.

Programming with Integers

In mathematics integers form an infinite set.

On a computer systems, integers are represented in binary.

The representation of an integer is a binary string of fixed length (precision), so there is only a finite number of “integers”.

Programming languages: signed & unsigned integers, short & long (& long long) integers, …

What Will Happen Here?

```
int i = 1;
while (i > 0)
{
i = i * 2;
}
```

Computing With Integers

Unsigned 8-bit integers

$255 + 1 = 0$ $16 * 17 = 16$

$0 - 1 = 255$

Signed 8-bit integers

$127 + 1 = -128$ $-128/-1 = -1$

In mathematics: $a + b \geq a$ for $b \geq 0$

As you can see, such obvious “facts” are no longer true.

Two's Complement

Signed integers are usually represented as 2's complement numbers.

The most significant bit (sign bit) indicates the sign of the integer:

- If sign bit is zero, the number is positive.
- If sign bit is one, the number is negative.

Positive numbers are given in normal binary representation.

Negative numbers are represented as the binary number that when added to a positive number of the same magnitude equals zero.

Two's Complement

Calculating the 2's complement representation of -*n*:

First, invert the binary equivalent of *n* by changing all ones to zeroes and all zeroes to ones:

- For 8-bit integers, this step computes 255-*n*

Then add one to the intermediate result:

- For 8-bit integers, this step computes 255-*n*+1= 256-*n*
- 256 corresponds to the carry bit.

	decimal	binary
	17	0001 0001
Step 1:	255-17	1110 1110
Step 2:	add 1	0000
	0001	
Result:	256-17	1110 1111

Integer Overflows

Integer overflows can lead to buffer overflows

Example (OS kernel system-call handler):

```
char buf[128];
combine(char *s1, size_t len1,
                  char *s2, size_t len2)
{
if (len1 + len2 + 1 <= sizeof(buf)) {
strncpy(buf, s1, len1);
strncat(buf, s2, len2);
}
}
```


Example from Markus Kuhn's lecture notes

Integer Overflows

The programmer has tried to check the string lengths to make a buffer overflow impossible.

Assume that **`len1 < sizeof(buf)`**.

On a 32-bit system, an attacker can set

```
len2 = 0xffffffff
```

and **`strncat`** will be executed because

```
len1 + 0xffffffff + 1 == len1
                      < sizeof(buf)
```

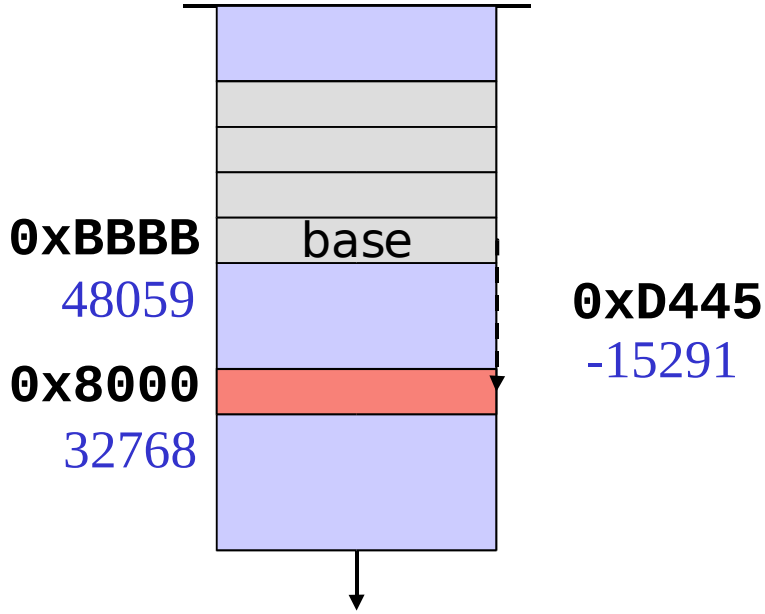
Array

You are given an array starting at memory location 0xBBBB (on a 16-bit machine)

Array elements are single words.

Which index do you write to so that memory location 0x8000 is overwritten?

You also must check lower bounds for array indices.



Canonicalization

Canonicalization: the process that determines how various equivalent forms of a name are resolved to a single standard name.

The single standard name is also known as the canonical name.

In general, an issue whenever an object has different but equivalent representations;

- Example: XML documents

Canonicalization must be idempotent.

Napster File Filtering

Napster was ordered by court to block access to certain songs.

Napster implemented a filter that blocked downloads based on the name of the song.

Napster users found a way around by using variations of the name of songs.

This is a particularly difficult problem because the users decide which names are equivalent.

Case-sensitive?

Security mechanism is case sensitive:

- MYFILE is different from MyFile

File system is case-insensitive:

- MYFILE is the same as MyFile

Permissions are defined for one version of the name only:

- Attacker requests access to another version.
- The security mechanism grants the request.
- The file system gives access to the resource that should have been protected.

Vulnerability in Apache web server with HFS+

Directory Traversal

An application may try to keep users in a specific directory.

Attack: walk out of the directory using ../; attack may try to hide ".." by using alternative UTF-8 encodings.

Relative file names: system starts from a list of predefined directories to look for the file.

Attack: put malicious code in a directory that is searched before the directory used by the application being attacked.

Don't filter for patterns, filter for results.

Variables

Buffer: concrete implementation of a variable.

If the value assigned to a variable exceeds the size of the allocated buffer, memory locations not allocated to this variable are overwritten.

If the memory location overwritten had been allocated to some other variable, the value of that other variable can be changed.

Depending on circumstances, an attacker could change the value of a protected variable A by assigning a deliberately malformed value to some other variable B.

Buffer overruns

Unintentional buffer overruns crash software, and have been a focus for reliability testing.

Intentional buffer overruns are a concern if an attacker can modify security relevant data.

Attractive targets are return addresses (specify the next piece of code to be executed) and security settings.

In languages like C or C++ the programmer allocates and de-allocates memory.

Type-safe languages like Java guarantee that memory management is ‘error-free’.

Buffer overrun (1980s)

Login in one version of Digital's VMS operating system: to log in to a particular machine, enter

username/DEVICE =<machine>

The length of the argument 'machine' was not checked; a device name of more than 132 bytes overwrote the privilege mask of the process started by login; users could thus set their own privileges.

System Stack

Function call: stack frame containing function arguments, return address, statically allocated buffers pushed on the stack.

When the call returns, execution continues at the return address specified.

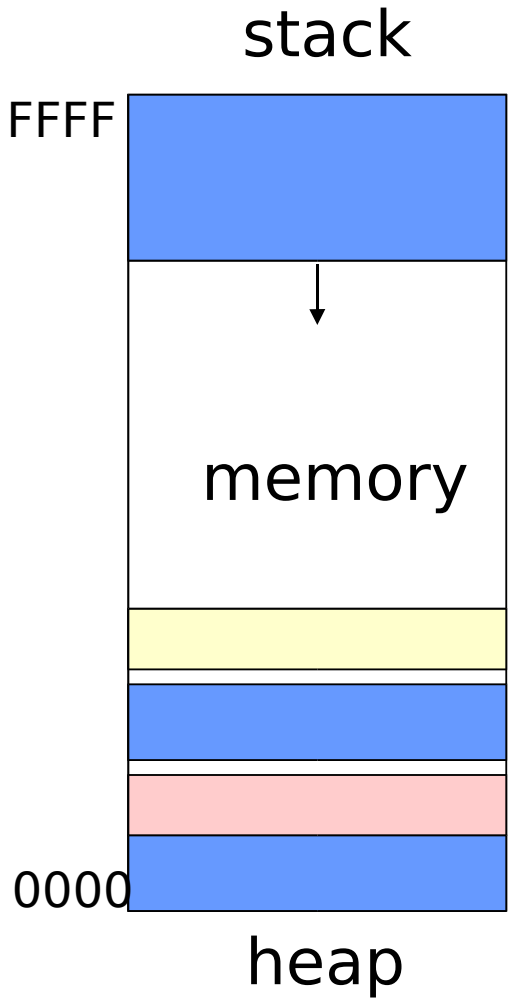
Stack usually starts at the top of memory and grows downwards.

Layout of stack frames is reasonably predictable.

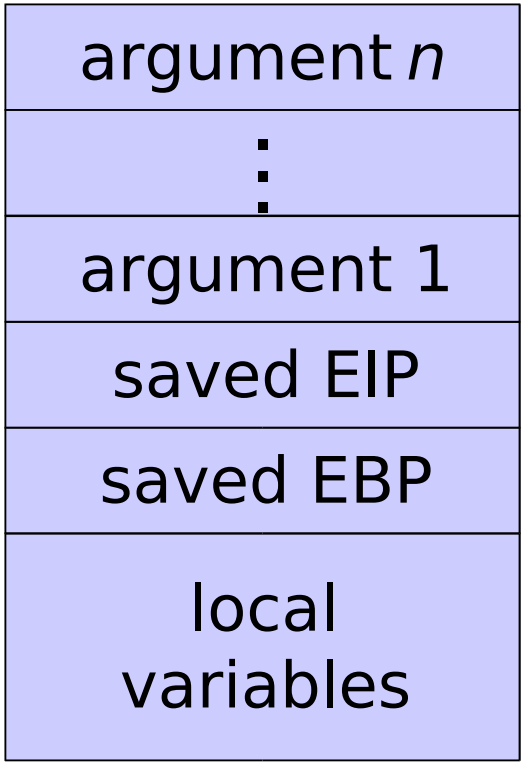
Stack & Heap

Stack: contains return address, local variables and function arguments; relatively easy to decide in advance where a particular buffer will be placed on the stack.

Heap: dynamically allocated memory; more difficult but by no means impossible to decide in advance where a particular buffer will be placed on the heap.



Stack Frame – Layout



extended instruction pointer (return address)

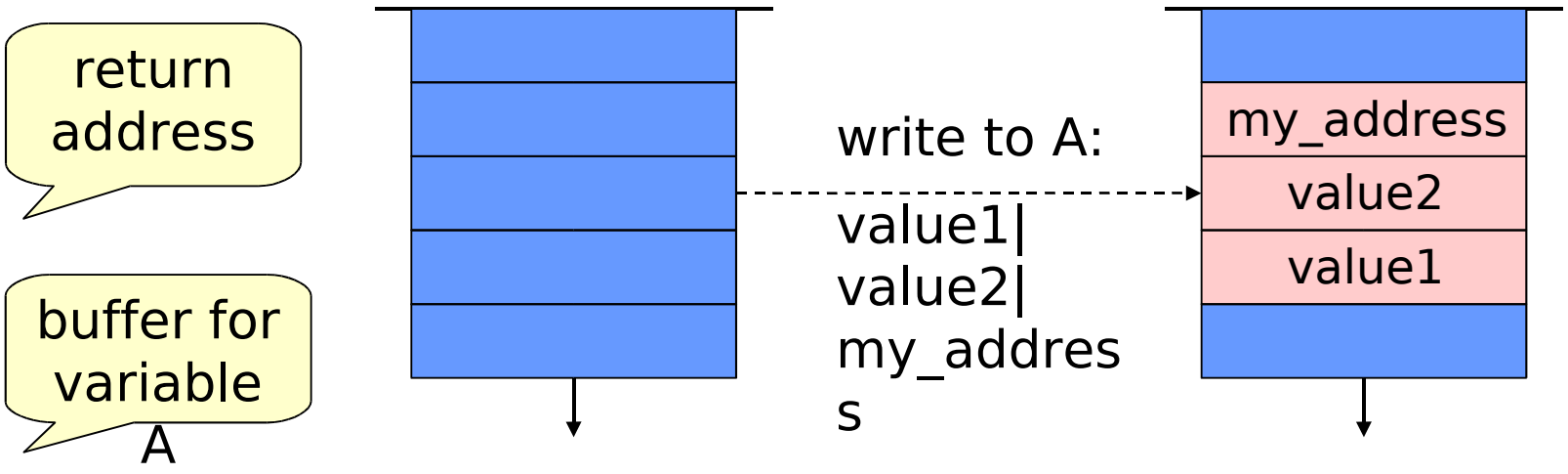
extended base pointer (reference point for relative addressing) a.k.a. frame pointer

Stack-based Overflows

Find a buffer on the runtime stack of a privileged program that can overflow the return address.

Overwrite the return address with the start address of the code you want to execute.

Your code is now privileged too.



Code Example

Declare a local short string variable

```
char buffer[80];
```

use the standard C library routine call

```
gets(buffer);
```

to read a single text line from standard input and save it into buffer.

Works fine for normal-length lines, but corrupts the stack if the input is longer than 79 characters.

Attacker loads malicious code into buffer and redirects return address to start of attack code.

How to Create an Exploit?

Use a specially crafted input to overwrite the return address and jump to the attack code.

Where to put the attack code ('shellcode')?

The shellcode could be put on the stack (as part of the malicious input).

To guess location, the attacker guesses the distance between return address and address of the input containing the shellcode.

Landing pad: NOP (no operation) instructions at start of shellcode to compensate for variations in the location the code is found.

Defence: Non-executable Stack

Stops attack code from being executed from the stack.

Memory management unit configured to disable code execution on the stack.

Not trivial to implement if existing O/S routines are executing code on the stack.

General issue – backwards compatibility: security measures may break existing code.

Attackers may find ways of circumventing this protection mechanism.

Detection – Compiler

Detect attempts at overwriting the return address.

Place a check value (‘canary’) in the memory location just below the return address.

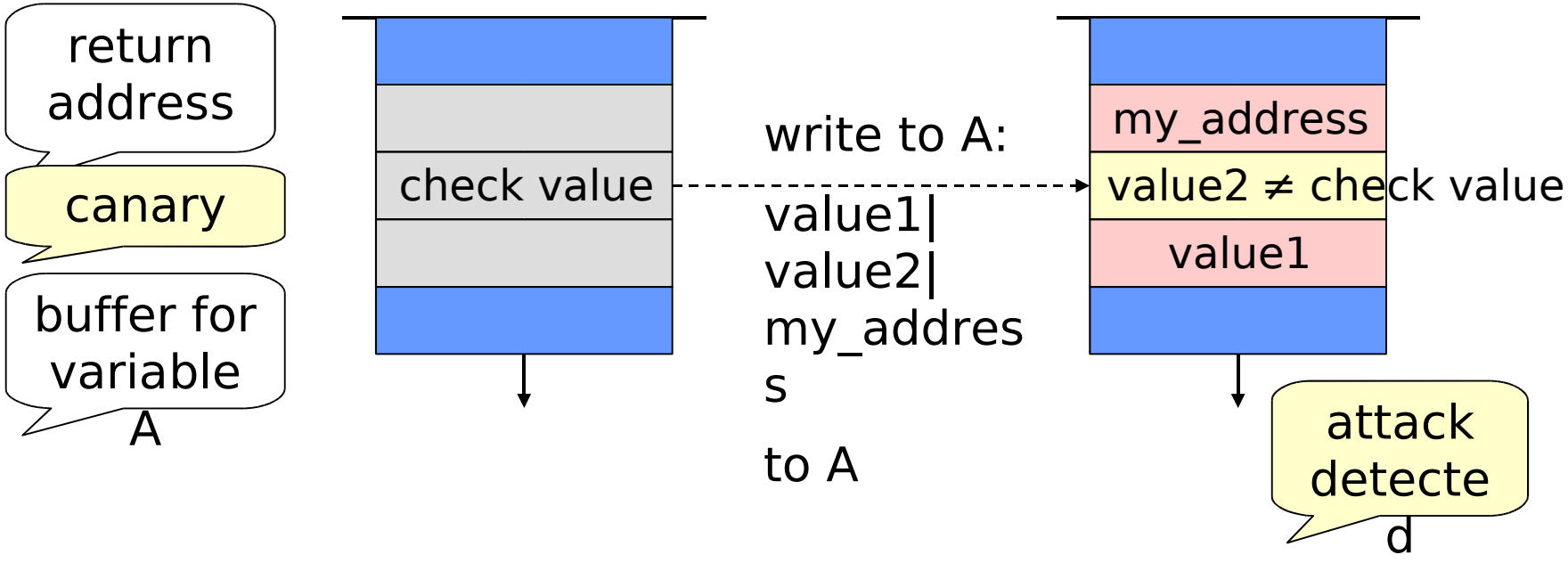
Before returning, check that the canary has not been changed.

Stackguard: random canaries.

- Alternatives: null canary, terminator canary

Source code has to be recompiled to insert placing and checking of the canary.

Canaries


return address
canary
buffer for variable A
check value
write to A:
value1|
value2|
my_addres
s
to A
my_address
value2 ≠ check value
value1
attack detecte
d

Heap Overruns

More difficult to determine how to overwrite a specific buffer.

More difficult to determine which other buffers will be overwritten in the process; if you are an attacker, you may not want to crash the system before you have taken over.

Even attacks that do not succeed all the time are a threat.

Can overwrite filenames and function pointers, and mess up memory management.

Type Safety – Java

Type safety (memory safety): programs cannot access memory in inappropriate ways.

Each Java object has a class; only certain operations are allowed to manipulate objects of that class.

Every object in memory is labelled with a class tag.

When a Java program has a reference to an object, it has internally a pointer to the memory address storing the object.

The pointer can be thought of as tagged with a type that says what kind of object the pointer is pointing to.

Type Confusion

Dynamic type checking: check the class tag when access is requested.

Static type checking: check all possible executions of the program to see whether a type violation could occur.

If there is a mistake in the type checking procedure, a malicious applet might be able to launch a type confusion attack by creating two pointers to the same object-with incompatible type tags.

Type Confusion

Assume the attacker manages to let two pointers point to the same location

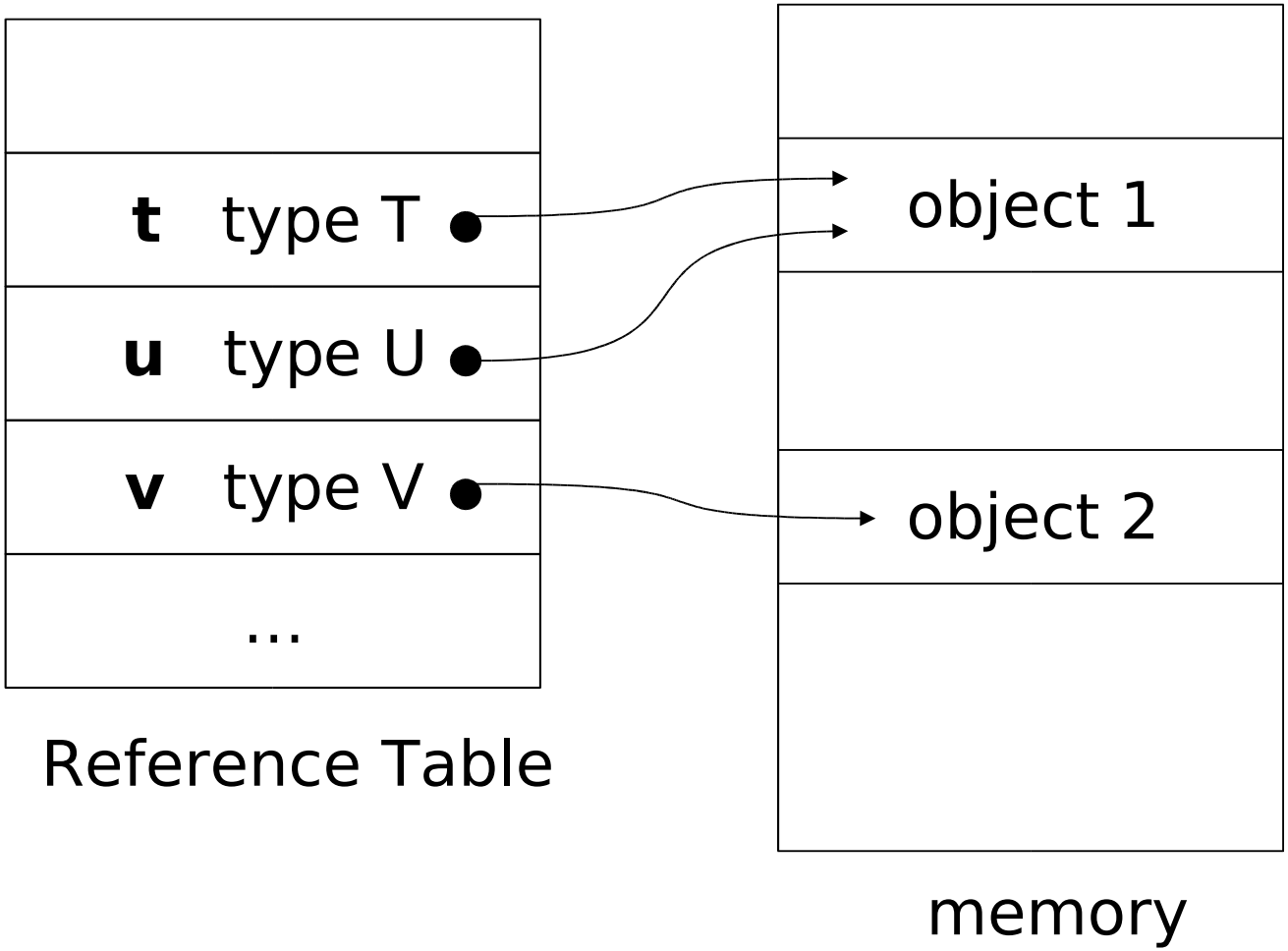
```
class T {
        SecurityManager x;
}

class U {
        MyObject x;
}
```

class definitions

```
T t = the pointer tagged T;
U u = the pointer tagged U;
t.x = System.getSecurity();
MyObject m = u.x;
```

malicious applet

Type Confusion



Type Confusion

The SecurityManager field can now also be manipulated from MyObject.

We sketch a type confusion attack in Netscape Navigator 3.0β5 (discovered by Drew Dean), fixed in version 3.0β6.

Source: Gary McGraw & Edward W. Felten: Java Security, John Wiley & Sons, 1997.

Netscape Vulnerability

Java allows a program that uses type T also to use type *array of* T.

Array types are defined by the VM for internal use; their name begins with the character [.

A programmer defined classname is not allowed to start with this character.

Hence, there should be no danger of conflict.

However, a Java byte code file could declare its own name to be a special array types name, thus redefining one of Java's array types.

Unix *rlogin*

Unix *login* command:

- **`login [[-p] [-h<host>] [[-f]<user>]`**
- **`-f`** option “forces” log in: user is not asked for password

Unix *rlogin* command for remote login:

- **`rlogin [-l<user>] <machine>`**
- The *rlogin* daemon sends a login request for <*user*> to <*machine*>

Attack (some versions of Linux, AIX):

- % rlogin -l -froot <machine>

Results in forced login as root at the designated machine

- % login -froot <machine>

Unix rlogin

Problem: Composition of two commands.

Each command on its own is not vulnerable.

However, *rlogin* does not check whether the "username" has special properties when passed to *login*.

This is a bit like the double decode problem with UTF8-encoded Unicode characters.

Scripting

In scripting languages, executables can be passed as arguments.

Example: A CGI script to send file to clientaddress:

cat file | **mail** clientaddress

With the “mail address” to@me | rm -rf / as input the server executes

cat file | **mail** to@me | rm -rf /

After mailing the file to@me, all files the script has permission to delete are deleted.

Unescaping

Escape characters: escape out of the current execution context:

‘Unescaping’: make input non-executable by commenting out escape characters.

For example, neutralize the escape character ‘;’ in “string1;string2” is by the comment ‘\...\’

“string1 \;string2\”

Escape characters are system specific.

SQL Inserts

Example query from SQL database:

string sql = "SELECT * FROM client WHERE name= '" + name + "' "

Intention: insert legal user name like 'Bob' into query.

Attack enter 'user name': Bob' OR 1=1 --

The SQL command becomes SELECT * FROM client WHERE name = Bob' OR 1=1--

Because 1=1 is TRUE, name = Bob OR 1=1 is TRUE, and the entire client database is selected; -- is a comment erasing anything that would follow.

Race conditions

Multiple computations access shared data in a way that their results depend on the sequence of accesses.

- Multiple processes accessing the same variable.
- Multiple threads in multi-threaded processes (as in Java servlets).

An attacker can try to change a value after it has been checked but before it is being used.

TOCTTOU (time-to-check-to-time-of use) is a well-known security issue.

Example – CTSS (1960s)

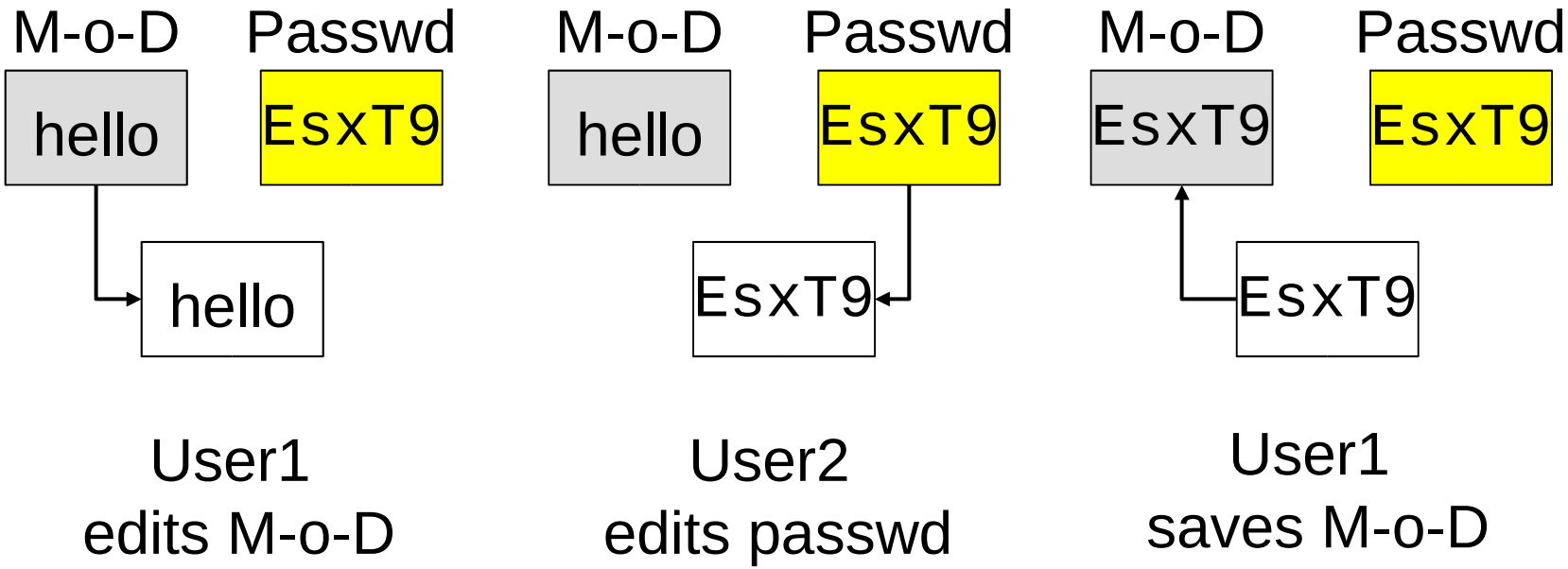
Password file shown as message of the day.

Every user had a unique home directory.

When a user invoked the editor, a scratch file with fixed name SCRATCH was created in this directory .

Innovation: Several users may work concurrently system manager.

Race Conditions



The abstraction ‘atomic transaction’ has been bro

Broken Abstractions

Treating the problems presented individually, would amount to penetrate-and-patch at a meta-level.

We looking for general patterns in insecure software, we see that familiar programming abstractions like variable, array, integer, data & code, address, or atomic transaction are being implemented in a way that breaks the abstraction.

Software security problems can be addressed

- in the processor architecture,
- in the programming language we are using,
- in the coding discipline we adhere to,
- through checks added at compile time (e.g. canaries),
- and during software development and deployment.

Prevention – Hardware

Hardware features can stop buffer overflow attacks from overwrite control information.

For example, a secure return address stack (SRAS) could protect the return address.

Separate register for the return address in Intel's Itanium processor.

With protection mechanisms at the hardware layer there is no need to rewrite or recompile programs; only some processor instructions have to be modified.

Drawback: existing software, e.g. code that uses multi-threading, may work no longer.

Prevention – Type Safety

Type safety guarantees absence of untrapped errors by static checks and by runtime checks.

The precise meaning of type safety depends on the definition of error.

Examples: Java, Ada, C#, Perl, Python, etc.

Languages needn't be typed to be safe: LISP

Type safety is difficult to prove completely.

Prevention – Safer Functions

C is infamous for its unsafe string handling functions: **`strcpy`**, **`sprintf`**, **`gets`**, …

Example: **`strcpy`**

```
char *strcpy( char *strDest,
const char *strSource );
```

- Exception if source or destination buffer are null.
- Undefined if strings are not null-terminated.
- No check whether the destination buffer is large enough.

Prevention – Safer Functions

Replace unsafe string functions by functions where the number of bytes/characters to be handled are specified:

`strncpy`, **`_snprintf`**, **`fgets`**, …

Example: **`strncpy`**

```
char *strncpy( char *strDest, const
char *strSource, size_t count );
```

You still have to get the byte count right.

- Easy if data structure used only within a function.
- More difficult for shared data structures.

Detection – Code Inspection

Code inspection is tedious: we need automation.

K. Ashcraft & D. Engler: Using Programmer-Written Compiler Extensions to Catch Security Holes, IEEE Symposium on Security &Privacy 2002.

Meta-compilation for C source code; ‘expert system’ incorporating rules for known issues: untrustworthy sources → sanitizing checks → trust sinks; raises alarm if untrustworthy input gets to sink without proper checks.

Code analysis to learn new design rules: Where is the sink that belongs to the check we see?

Microsoft has internal code inspection tools.

Detection – Testing

White box testing: tester has access to source code.

Black-box testing when source code is not available.

You do not need source code to observe how memory is used or to test how inputs are checked.

Example: syntax testing of protocols based on formal interface specification, valid cases, anomalies.

Applied to SNMP implementations: vulnerabilities in trap handling and request handling found http://www.cert.org/advisories/CA-2002-03.html

- Found by Oulu University Secure Programming Group http://www.ee.oulu.fi/research/ouspg/

Mitigation – Least Privilege

Limit privileges required to run code; if code running with few privileges is compromised, the damage is limited.

Do not give users more access rights than necessary; do not activate options not needed.

Example – debug option in Unix sendmail: when switched on at the destination, mail messages can contain commands that will be executed on the destination system.

Useful for system managers but need not be switched on all the time; exploited by the Internet Worm of 1988.

In the past, software was shipped in open configurations (generous access permissions, all features activated); users had to harden their systems by removing features and restricting access rights.

Today, software often shipped in locked-down configurations; users have to activate the features they want to use.

Reaction – Keeping Up-to-date

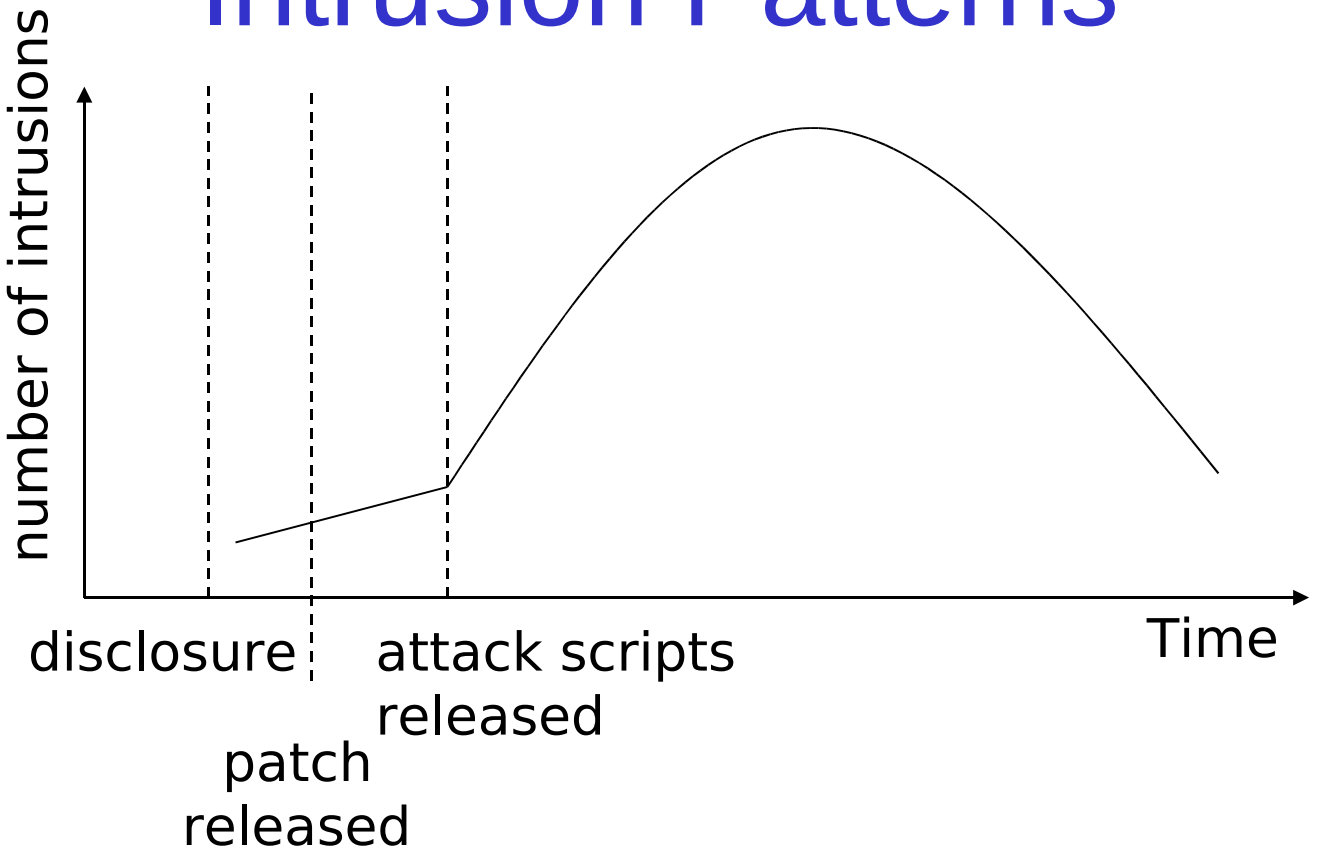
Information sources : CERT advisories, BugTraq at www.securityfocus.com, security bulletins from software vendors.

Hacking tools use attack scripts that automatically search for and exploit known type of vulnerabilities.

Analysis tools following the same ideas will cover most real attacks.

Patching vulnerable systems is not easy: you have to get the patches to the users and avoid introducing new vulnerabilities through the patches.

Intrusion Patterns



W. Arbaugh, B. Fithen, J. McHugh: Windows of Vulnerability:
A Case Study Analysis, IEEE Computer, 12/2000

Summary

Many of the problems listed may look trivial.

There is no silver bullet:

- Code-inspection: better at catching known problems, may raise false alarms.
- Black-box testing: better at catching known problems.
- Type safety: guarantees from an abstract (partial) model need not carry over to the real system.

Experience in high-level programming languages may be a disadvantage when writing low level network routines.