Secure Programming

A.A. 2022/2023

Corso di Laurea in Ingegneria delle Telecomnicazioni

B. Build Security In

Paolo Ottolino

Politecnico di Bari

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

Secure Programming Lab: Course Program

A. **Intro Secure Programming: «Who-What-Why-When-Where-How»**
B. **Building Security in: Buffer Overflow, UAF, Command Inection**
C. **Architecture and Processes: App Infrastructure, Three-Tiers, Cloud, Containers, Orchestration**
D. **SwA (Software Assurance): Vulnerabilities and Weaknesses (CVE, OWASP, CWE)**
E. **Security & Protection: Risks, Attacks. CIA -> AAA (AuthN, AuthZ, Accounting) -> IAM, SIEM, SOAR**
F. **Architecture and Processes 2: Ciclo di Vita del SW (SDLC), DevSecOps**
G. **Dynamic Security Test: VA, PT, DAST (cfr. VulnScanTools), WebApp Sec Scan Framework (Arachni, SCNR)**
H. **Free Security Tools: OWASP (ZAP, ESAPI, etc), NIST (SAMATE, SARD, SCSA, etc), SonarCube, Jenkins**
I. **Architecture and Processes 3: OWASP DSOMM, NIST SSDF**
J. **Operating Environment: Kali Linux on WSL**
K. **Python: Powerful Language for easy creation of hacking tools**
L. **SAST: Endogen, Exogen factors, SAST (cfr. SourceCodeAnalysisTools), SonarQube**
M. **Exercises: SecureFlag**

Build Security In: Agenda

1. **Security In**: What is?
2. **BOF**: Buffer OverFlow (pointer concepts)
3. **UAF**: Use After Free
4. **Unsecured Input**: Command Injection (provide data as code)
5. **Secure Coding Practice**: SEI (Software Engineering Institute)

B.1 Security In: What is?

The 15 biggest data breaches of the 21st century

#	Company	Date	Accounts	Details
1	Yahoo	Augt 2013	3 billions	Incident announced in 2016. Reduce prize in Verizon acquisition
2	Aadhaar	Jan 2018	1.1 billion	Indian citizens' identity/biometric information exposed
3	Alibaba	Nov 2019	1.1 billion	Pieces of user data (including usernames and mobile numbers)
4	LinkedIn	June 2021	700 millions	User data posted in on a dark web forum
5	Sina Weibo	Mar 2020	538 millions	Data (real, site, gender, location, mobile) of user of the Social Media
6	Facebook	Apr 2019	533 millions	2 datasets exposed on public Internet (HIBP: HaveIBeenPwned)
7	Marriott	Sep 2018	500 millions	Exposure of sensitive details about customers
8	Yahoo	2014	500 millions	User data (names, email addresses, phone numbers, hashed passwords, and dates of birth)
9	Adult Friend Finder	Oct 2016	412.2 millions	20 years' worth of user data across six databases
10	MySpace	2013	360 millions	accounts leaked onto LeakedSource.com and put up for sale on dark web market The Real Deal
11	NetEase	Oct 2015	235 millions	email addresses and plaintext passwords of the email accounts sold by dark web
12	Experian (Court Ventures)	Oct 2013	200 million	Vietnamese man (Hieu Minh Ngo) posing as a private investigator of Singapore got access to DB (about 2$ million revenue)
13	LinkedIn	June 2012	165 millions	Revealed only in 2016. Perpetrated by the same hacker of MySpace
14	Dubsmash	Dec 2018	162 millions	Personal data (email, username, PBKDF2 password hashes, birth etc.) of the video messaging service put up for sale on the Dream Market dark web market
15	Adobe	Oct 2013	153 millions	encrypted customer credit card records and login data

https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html

B.1b Security In: What is?

Vulnerabilities: Security-relevant Defects

The causes of security breaches are varied, but many of them owe to a **defect** (or bug) or **design flaw** in a targeted computer system's software.

- Software defect (bug) or design flaw can be **exploited** to affect an undesired behavior

B.1b Security In: What is?

Defects and Vulnerabilities

The use of software is growing ➔ So: more bugs and flaws

Software is large (lines of code)

- Chevy volt: 10 million
- Boeing 787: 14 million
- F35 fighter Jet: 24 million
- Windows: 50 million
- Mac OS: 80 million
- Google: 2 billion

B.1c Security In: What is?

Quiz 1

Program testing can show that a program has no bugs.

A. True

B. False

B.1d Security In: What is?

Quiz 1

Program testing can show that a program has no bugs.

A. True

B. False

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger Dijkstra

B.1e Security In: What is?

Considering Correctness

- All software is buggy, isn’t it? Haven’t we been dealing with this for a long time?
- Removing bugs is expensive
- A normal user never sees most bugs, or figures out how to work around them
- Therefore, companies fix the most likely bugs, to save money

B.1f Security In: What is?

Exploitable Bugs

Many kinds of exploits have been developed over time, with technical names like:

- Buffer overflow
- Use after free
- Command injection
- SQL injection
- Privilege escalation
- Cross-site scripting
- Path traversal
- …

B.1g Security In: What is?

Agenda

- The basics of threat modeling.

- Three basic kinds of exploits:

1. Buffer Overflows → Type-safe Programming Languages
2. Use After Free → Type-safe Programming Languages
3. Command injection → Input Validation.

B.2 Buffer Overflow: pointer concepts

outside its allotted bounds

A buffer overflow describes a <u>family of possible exploits</u> of a vulnerability in which a program may incorrectly access a buffer outside its allotted bounds.

- A buffer overwrite occurs when the <u>out-of-bounds</u> access is a write.
- A buffer overread occurs when the <u>out-of-bounds</u> access is a read.

B.2b Buffer Overflow: pointer concepts

Example: Out-of-Bounds in C

```c
#include <stdio.h>

void incr_arr(int *x, int len, int i) {
  if (i >= 0 && i < len) {
    x[i] = x[i] + 1;
    incr_arr(x,len,i+1);
  }
}

int y[10] = {1,1,1,1,1,1,1,1,1,1};
int z = 20;

int main(int argc, char **argv) {
  incr_arr(y,11,0);
  printf("%d =? 20\n",z);
  return 0;
}
```

Output:

```
21 =? 20
```

The value of z changed from 20 to 21. Why?

B.2c Buffer Overflow: pointer concepts

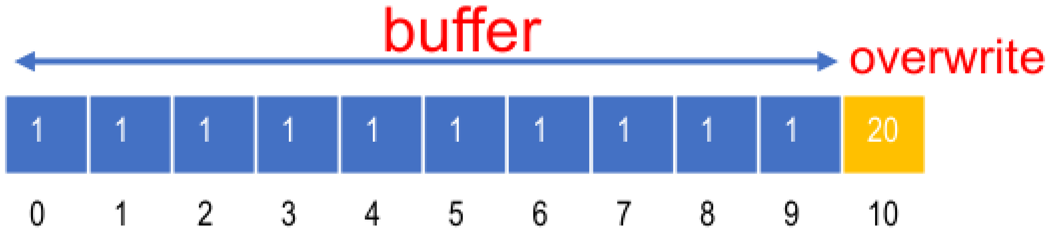
Example: Out-of-Bounds in C

```c
1  #include <stdio.h>
2
3  void incr_arr(int *x, int len, int i) {
4    if (i >= 0 && i < len) {
5      x[i] = x[i] + 1;
6      incr_arr(x,len,i+1);
7    }
8  }
9
10 int y[10] = {1,1,1,1,1,1,1,1,1,1};
11 int z = 20;
12
13 int main(int argc, char **argv) {
14   incr_arr(y,11,0);
15   printf("%d =? 20\n",z);
16   return 0;
17 }
```

Output:

```
21 =? 20
```

- array y has length 10
- but the second argument of incr_arr is 11, which is one more than it should be.
- As a result, line 5 will be allowed to read/write past the end of the array.



B.2d Buffer Overflow: pointer concepts

Example: Out-of-Bounds in type-safe language

```
let rec incr_arr x i len =
  if i >= 0 && i < len then
    (x.(i) <- x.(i) + 1;
    incr_arr x (i+1) len)
;;

let y = Array.make 10 1;;
incr_arr y 0 (1 + Array.length y);;
```

Consider the same program, written in Type-safe language

- Exception: Invalid_argument "index out of bounds".

- type-safe language detects the attempt to write one past the end of the array and signals by throwing an exception.

B.2e Buffer Overflow: pointer concepts

Exploiting a Vulnerability


```
int y[10]={1,1,1,1,1,1,1,1,1,1};
int z = 20;
```

```
#include <stdlib.h>
int main(int argc, char **argv) {
  int len = 10;
  if (argc == 2) len = atoi(argv[1]);
  incr_arr(y,len,0);
  printf("%d =? 20\n",z);
  return 0;
}
```

`a.out`

`a.out 11`

If an attacker can force the argument to be 11 (or more), then he can trigger the bug.

B.2f Buffer Overflow: pointer concepts

Quiz 2

If you declare an array as `int a[100];` in C and you try to write `5` to `a[i]`, where i happens to be `200`, what will happen?

A. Nothing

B. The C compiler will give you an error and won't compile

C. There will always be a runtime error

D. Whatever is at `a[200]` will be overwritten

B.2g Buffer Overflow: pointer concepts

Quiz 2

If you declare an array as `int a[100];` in C and you try to write `5` to `a[i]`, where i happens to be `200`, what will happen?

A. Nothing

B. The C compiler will give you an error and won’t compile

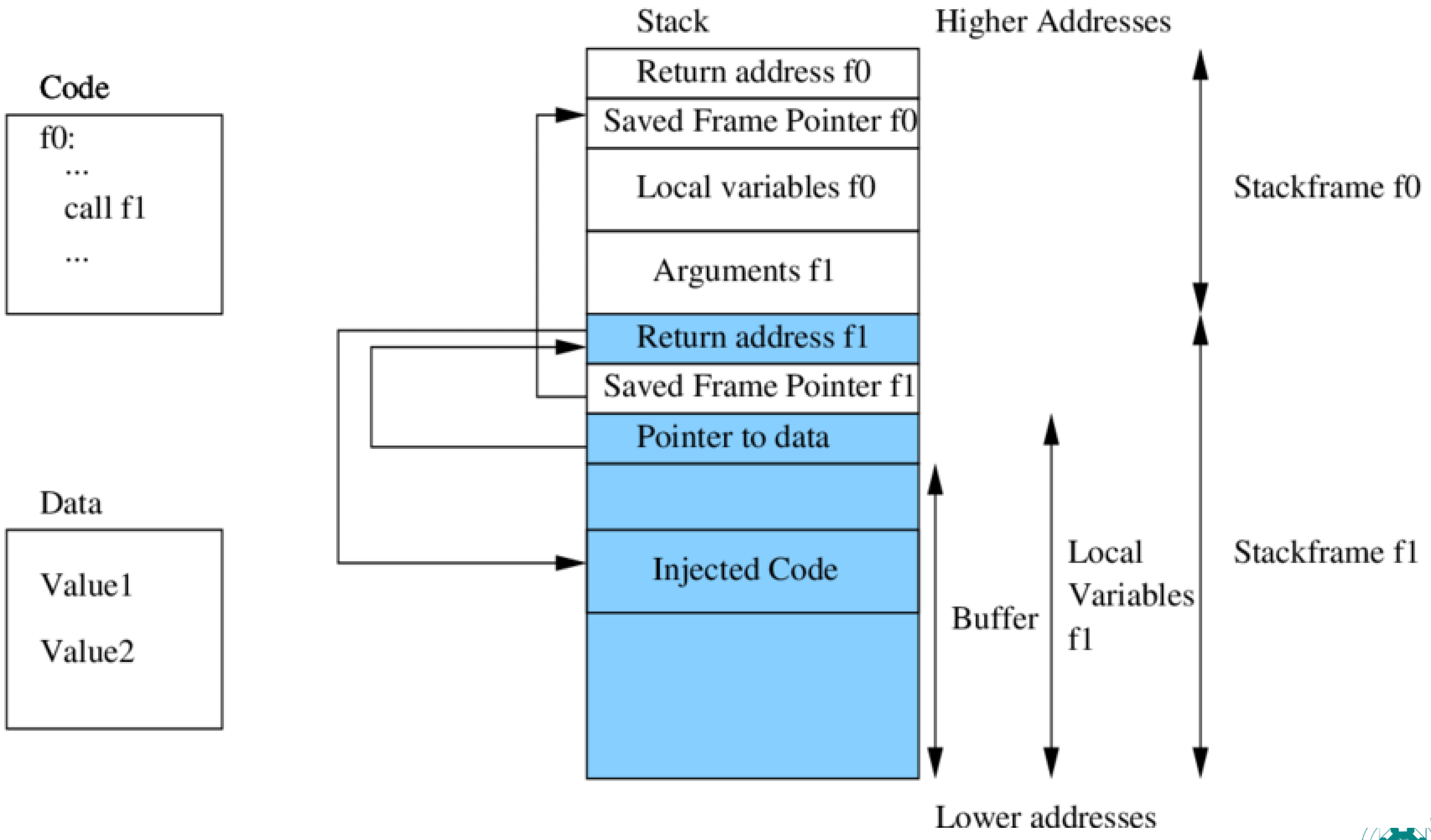
C. There will always be a runtime error

D. Whatever is at `a[200]` will be overwritten

What Can Exploitation Achieve? Heartbleed

- Heartbleed is a bug in the popular, open-source OpenSSL codebase, part of the HTTPS protocol.
- The attacker can read the memory beyond the buffer, which could contain secret keys or passwords, perhaps provided by previous clients

What Can Exploitation Achieve? Morris Worm



Morris Worm

- For C/C++ programs: A buffer with the password could be a local variable
- Therefore: The attacker’s input (includes machine instructions) is too long, and overruns the buffer
- The overrun rewrites the return address to point into the buffer, at the machine instructions
- When the call “returns” it executes the attacker’s code

B.2j Buffer Overflow: pointer concepts

Quiz 3

Which kinds of operation is most likely to not lead to a buffer overflow in C?

A. Floating point addition

B. Indexing of arrays

C. Dereferencing a pointer

D. Pointer arithmetic

B.2k Buffer Overflow: pointer concepts

Quiz 3

Which kinds of operation is most likely to not lead to a buffer overflow in C?

A. Floating point addition

B. Indexing of arrays

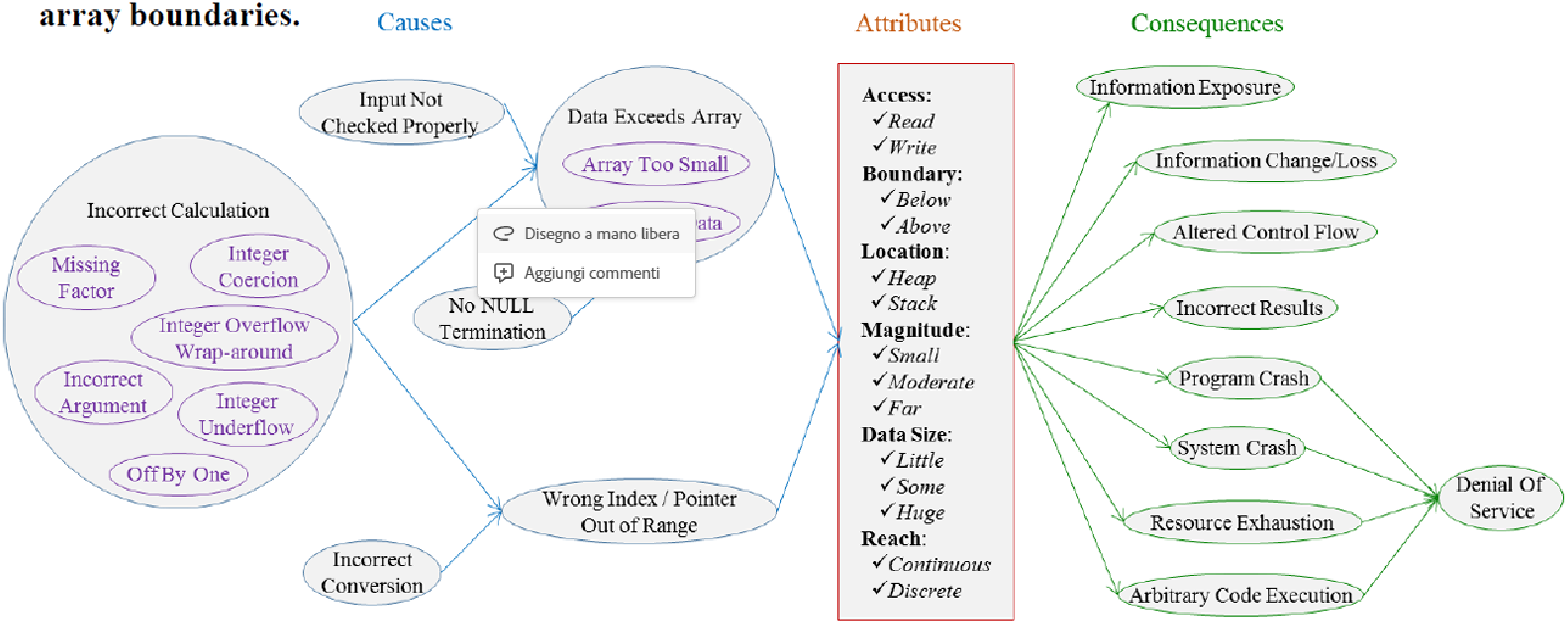
C. Dereferencing a pointer

D. Pointer arithmetic

B.2l Buffer Overflow: pointer concepts

Causes

Buffer Overflow (BOF): The software can access through an array a memory location that is outside the array boundaries.

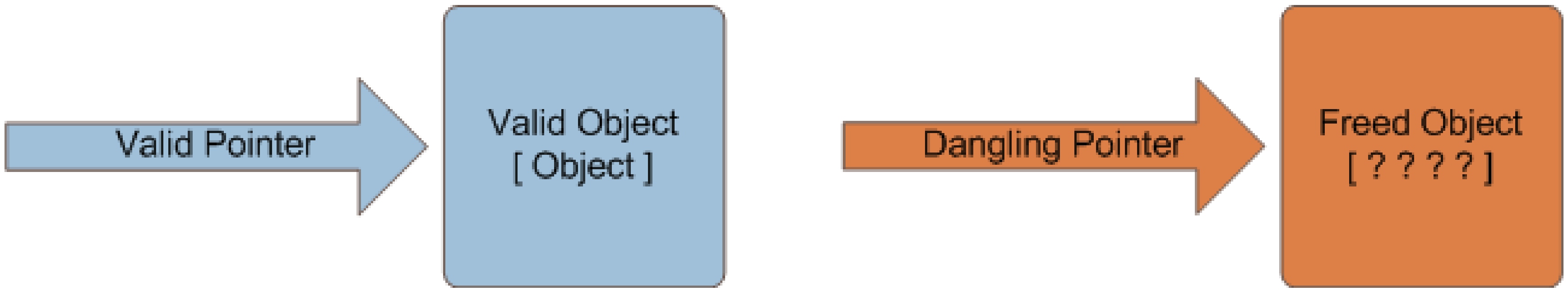


Source: Bojanova, et al, "The Bugs Framework (BF): A Structured, Integrated Framework to Express Software Bugs",2016, http://www.mys5.org/Proceedings/2016/Posters/2016-S5-Posters_Wu.pdf

B.3 User After Free

Definition (bug, no exploit)

- Use-after-free referencing stale data…



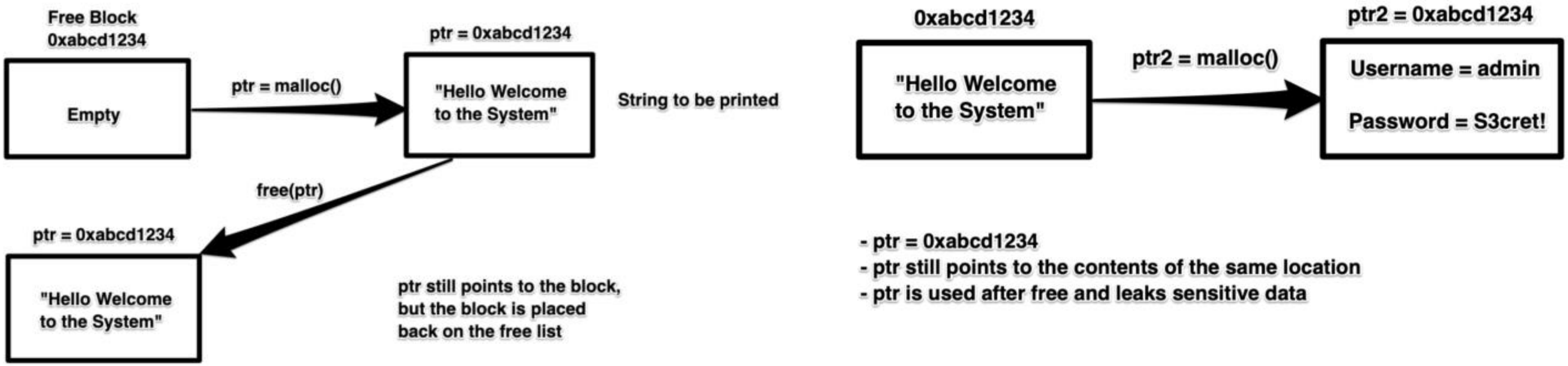
If this code is executed and if the error branch is taken, an undefined behavior is likely to occur since ptr points to a non-valid memory area

```
`char * ptr = malloc(SIZE);
…
if (error){
 free(ptr);
}
…
printf("%s", ptr);
```

B.3 User After Free

Definition (bug, no exploit)

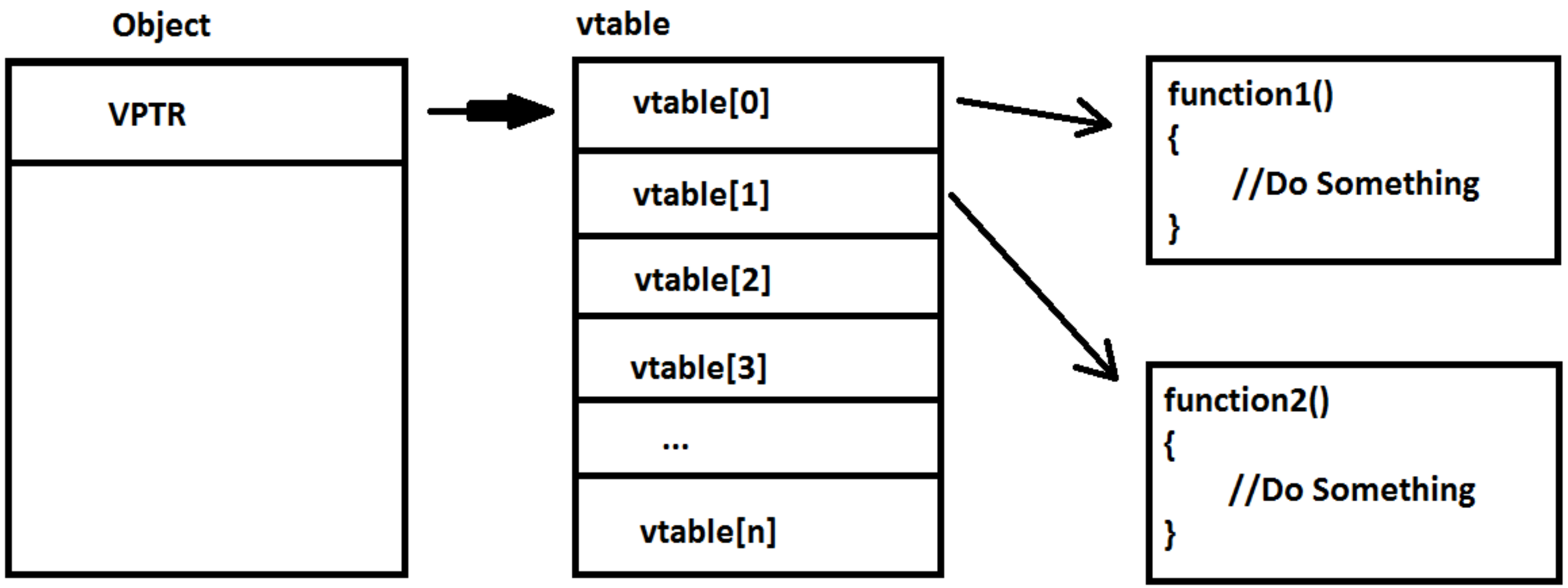
- Use-after-free can cause access to sensitive data…



B.3 User After Free

Definition (bug, no exploit)

- Use-after-free can cause stale data to be treated as code



C and C++ programs expect the programmer to ensure this never happens!

B.3 User After Free

Definition (bug, no exploit)

- Use-after-free relies on the ability to keep using freed memory once it’s been reallocated
- Buffer overflows rely on the ability to read or write outside the bounds of a buffer

```c
#include <stdlib.h>
struct list {
  int v;
  struct list *next;
};
int main() {
  struct list *p = malloc(sizeof(struct list));
  p->v = 0;
  p->next = 0;
  free(p); // deallocates p
  int *x = malloc(sizeof(int)*2); // reuses p's old memory
  x[0] = 5; // overwrites p->v
  x[1] = 5; // overwrites p->next
  p = p->next; // p is now bogus
  p->v = 2; // CRASH!
  return 0;
}
```

C and C++ programs expect the programmer to ensure this never happens!

B.3b User After Free

Type-safe language

Defense: Type-safe Languages

Type-safe Languages (like Python, Java, etc.) ensure buffer sizes are respected

• Compiler inserts checks at reads/writes. Such checks can halt the program. But will prevent a bug from being exploited

• Garbage collection avoids the use-after-free bugs. No object will be freed if it could be used again in the future.

B.3c User After Free

Type-safe Language

Defense: Type-safe Languages

Type safety ensures two useful properties that preclude buffer overflows and other memory corruption-based exploits.

- Preservation: memory in use by the program at a particular type T always has that type T.
- Progress: values deemed to have type T will be usable by code expecting to receive a value of that type
- To ensure preservation and progress implies that only non-freed buffers can only be accessed within their allotted bounds, precluding buffer overflows.
- Overwrites breaks preservation
- Overreads could break progress
- Uses-after-free could break both

B.3d User After Free

Type-safe Languages

Type safety

Informally, a type-safe language is one for which:

- There is a clearly specified notion of **type correctness**.
- Type correct programs are **free** of "runtime type errors".

B.3e User After Free

Type-safe Languages

Type safety

Type safety is a matter of **coherence** between the **static** and **dynamic semantics**.

- The **static** semantics makes **predictions** about the execution behavior.

- The **dynamic** semantics must **comply** with those predictions.

B.3e2 User After Free

Type-safe Languages

Type safety

Examples

1. if the **type system** tracks **sizes of arrays**, then **out-of-bounds** subscript is a run-time type error**.**
   - The type system ensures that **access** is within **allowable limits**.
   - If the run-time model exceeds these bounds, you have a run-time type error.

2. Similarly, if the **type system** tracks **value ranges**, then **division by zero** or **arithmetic overflow** is a run-time type error.

B.3f User After Free

Use After Free (bug, no exploit)

Type-safe Languages: Costs

- **Performance**

Array Bounds Checks and Garbage Collection add overhead to a
➔ program's running time.

- **Expressiveness**

C casts between different sorts of objects, e.g., a struct and an array.
This sort of operation -- cast from integer to pointer -- is not permitted in a type safe language.
➔ Need casting in System programming

B.3g User After Free

Quiz 4

Applications developed in the programming languages ____________ are susceptible to buffer overflows and uses-after-free.

A. Ruby, Python

B. Java, Pascal

C. C, C++

D. Rust, C#

B.3h User After Free

Quiz 4

Applications developed in the programming languages ____________ are susceptible to buffer overflows and uses-after-free.

A. Ruby, Python

B. Java, Pascal

C. C, C++

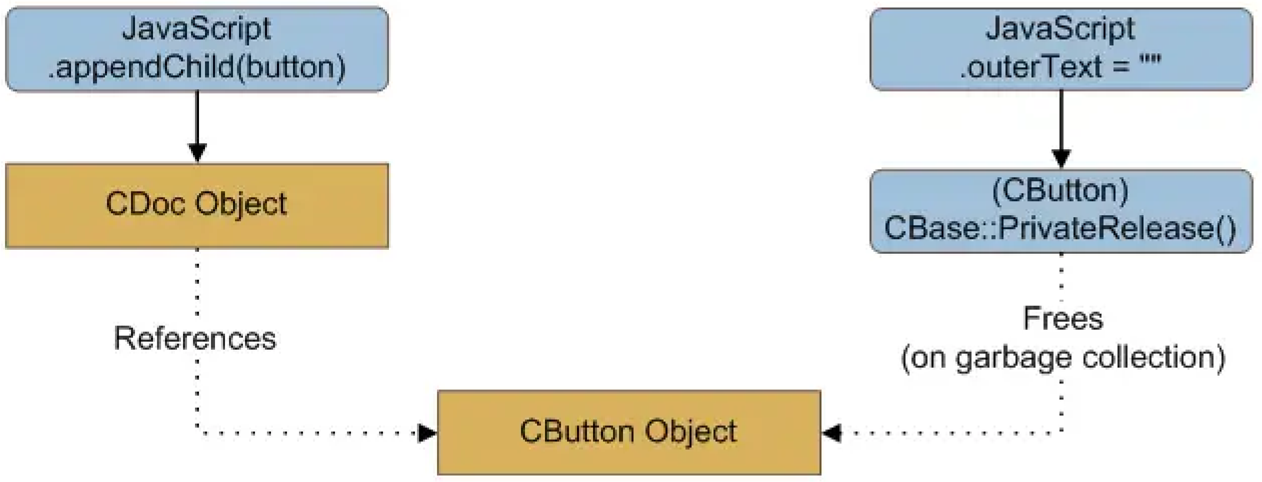
D. Rust, C#

B.3i User After Free

Internet Explorer: many UAF

Use-after-free is still a common bug class because the task of manually identifying them, especially in large and complex codebases is a challenge.

The reason is that their existence is a result of the combined actions from different parts of an application

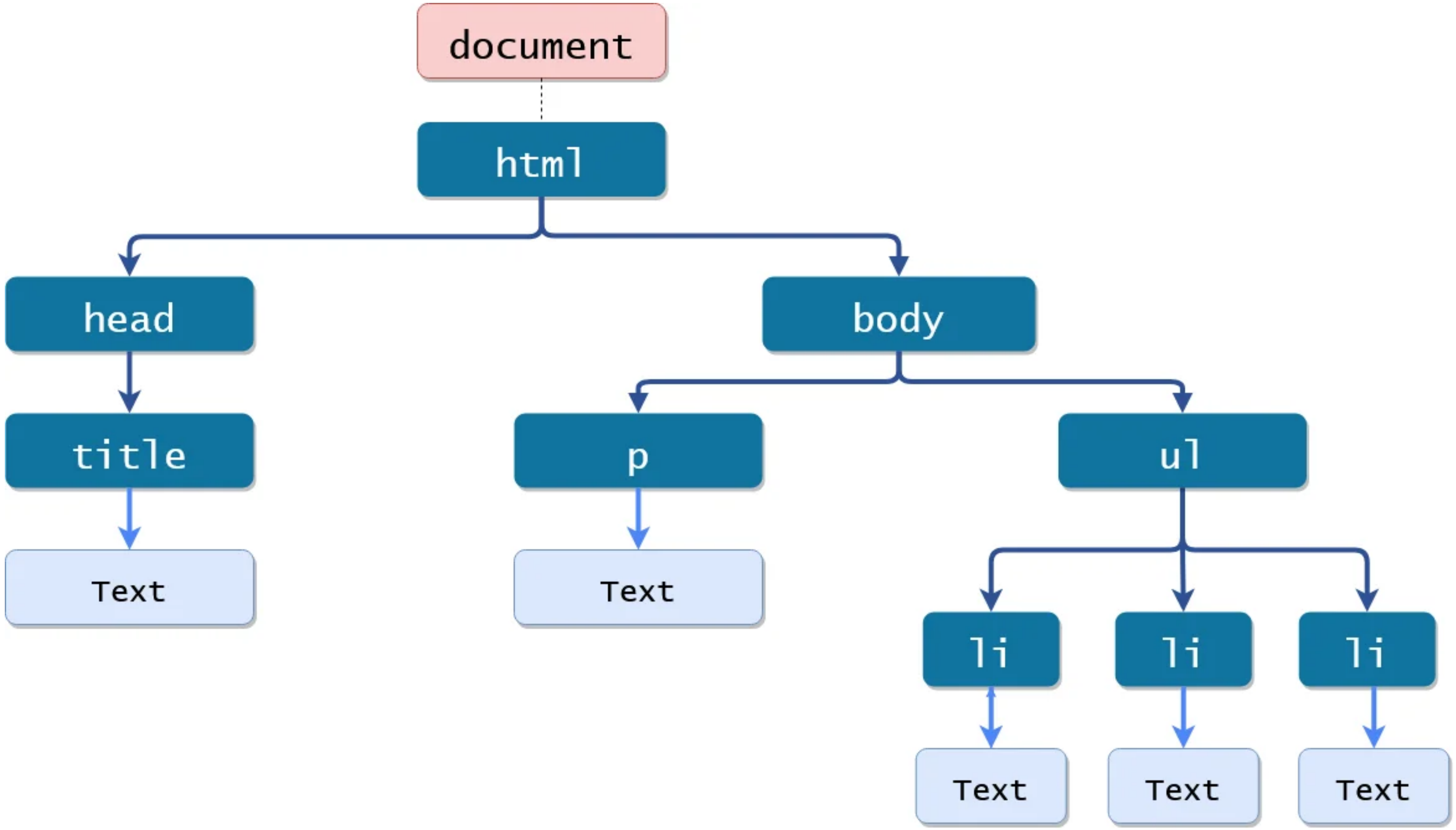


https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/

B.3i2 User After Free

Internet Explorer: page rendering → DOM

DOM: The **Document Object Model** (*DOM*) is the data representation of the objects that comprise the structure and content of a document on the web. This guide will introduce the DOM, look at how the DOM represents an [HTML](#) document in memory and how to use APIs to create web content and applications.



https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/

B.3i2 User After Free

Internet Explorer: DOM → JS

Nodes in HTML DOM are accessed by using javascript.

There are many DOM access methods using which you can access HTML elements:

- getElementById(): returns an element whose id is matched with the passed id value within the method;

- getElementsByClassName(): returns an array of all the child elements which have given class name(s);

- getElementsByTagName(): returns an array of all HTML elements with the given tag name in form of an array
- querySelector(): selects the 1st element on the basic of a valid CSS selectors string
- querySelectorAll(): selects all the element matching the string and return as a collection

https://www.tutorialstonight.com/js/js-dom-access-methods

B.3i3 User After Free

Internet Explorer: JS → JIT Compiler

JIT (Just in Time) Compiler is not Type-Safe → UAF

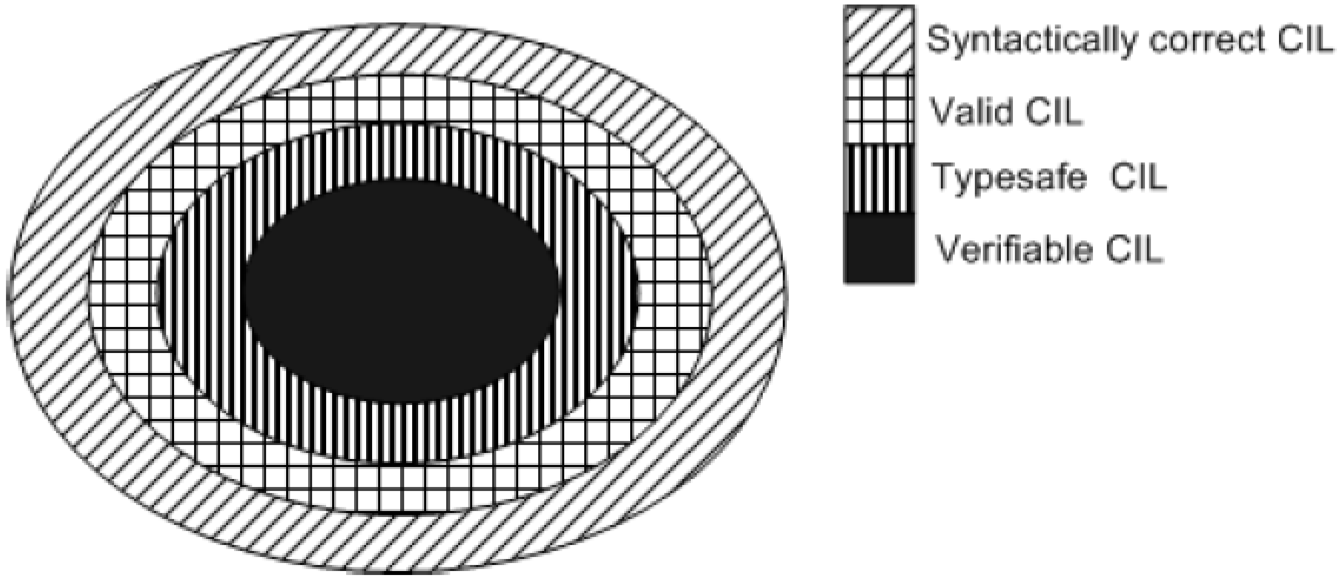


Figure 1: Relationship between correct and verifiable CIL

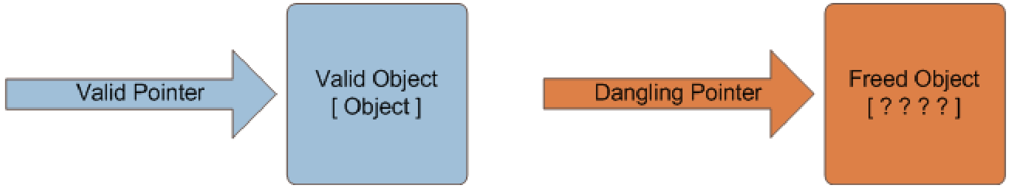
CIL: Common Intermediate Language

https://tooslowexception.com/net-jit-compiler-is-not-type-safe/

https://www.geeksforgeeks.org/what-is-just-in-time-jit-compiler-in-dot-net/

B.3i4 User After Free

Internet Explorer: many UAF



Two common reasons that lead to dangling pointers are:

•**Not updating** a **pointer value** once the **object** it points to is **freed**.

•**Not updating** the **reference count** of a currently **in-use object**. This results in the object currently in-use to be prematurely freed.

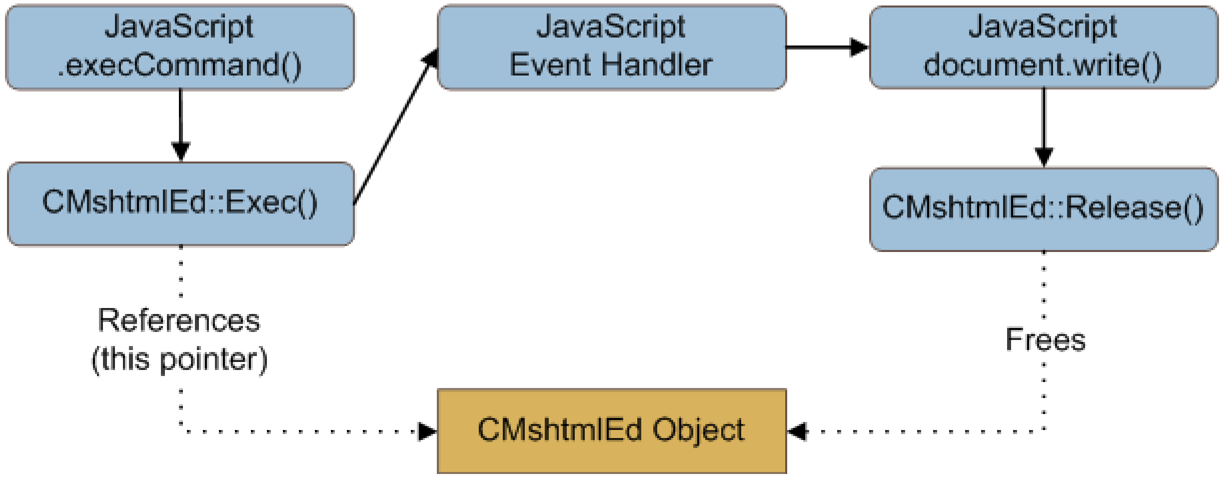
Typically, exploits that leverage UAFs will attempt to reallocate the memory previously allocated to the freed object. This causes the dangling pointer to point to an attacker-controlled data. The application's execution flow is then controlled when an attacker-controlled data obtained via the dangling pointer is used within the application.

https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/

B.3i4 User After Free

Internet Explorer: many UAF

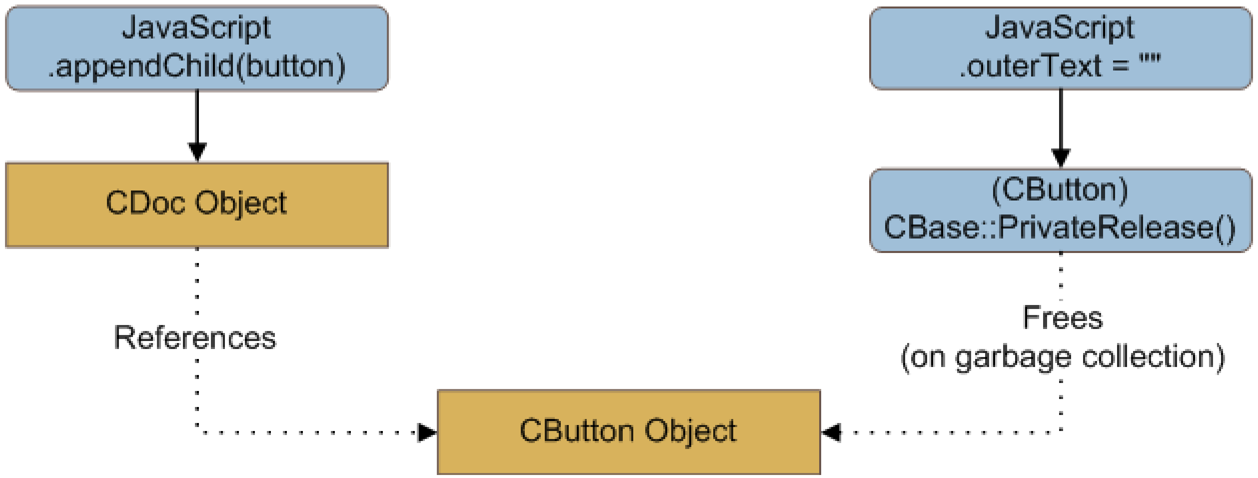
Example 1: CVE-2012-4969 (IE CMshtmlEd UAF)



https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/

B.3i5 User After Free

Internet Explorer: many UAF



Example 2: CVE-2012-4792 (IE CButton UAF)

https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/

B.3j User After Free

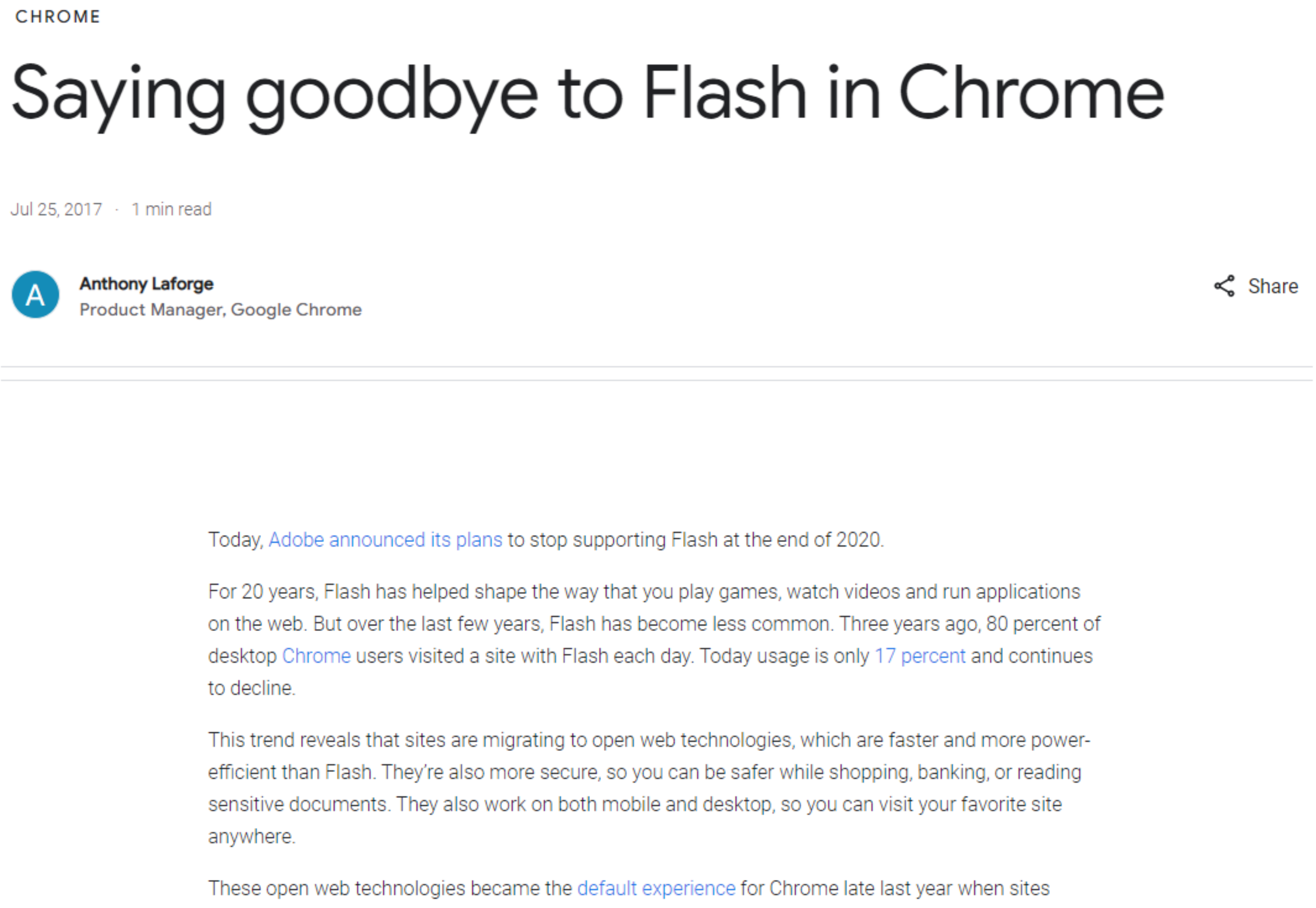
Adobe Flash Player

The classic Flash exploit for many past years was mainly about corruption of the length field of Vector objects.

The length field is located at the beginning of the Vector buffer.

For instance, a heap overflow exploit sprays Vectors and creates memory holes by freeing some Vectors.

Vulnerable buffer is created to occupy one of the memory holes, corrupting the length field of a Vector object by triggering an overflow.



CHROME

Saying goodbye to Flash in Chrome

Jul 25, 2017 · 1 min read

A **Anthony Laforge**
Product Manager, Google Chrome

Share

Today, Adobe announced its plans to stop supporting Flash at the end of 2020.

For 20 years, Flash has helped shape the way that you play games, watch videos and run applications on the web. But over the last few years, Flash has become less common. Three years ago, 80 percent of desktop Chrome users visited a site with Flash each day. Today usage is only 17 percent and continues to decline.

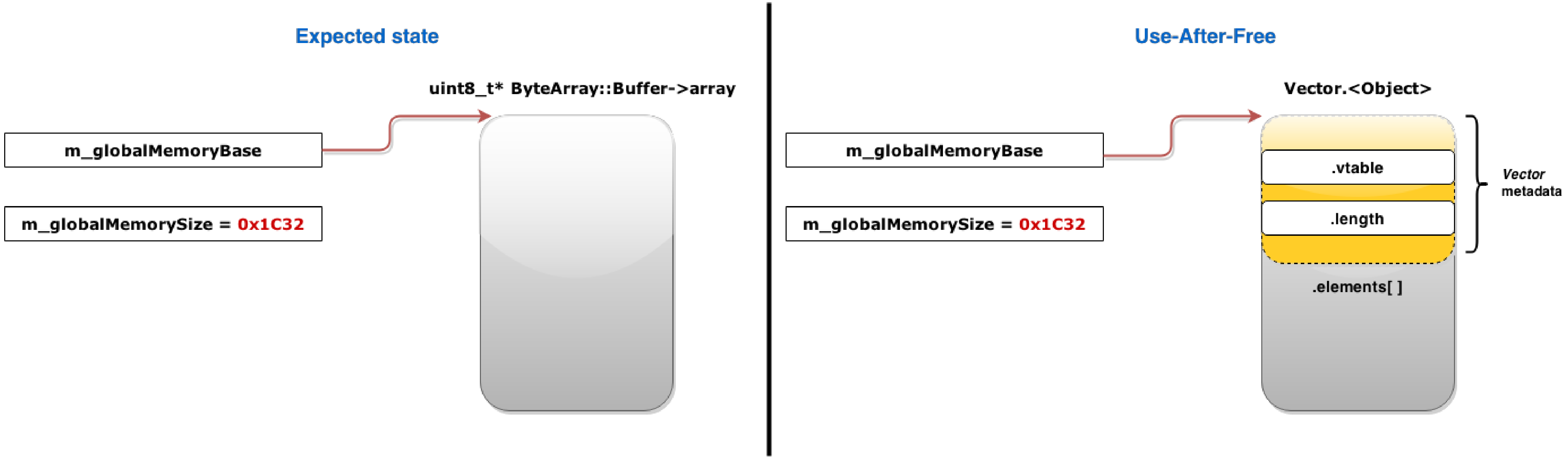
This trend reveals that sites are migrating to open web technologies, which are faster and more power-efficient than Flash. They're also more secure, so you can be safer while shopping, banking, or reading sensitive documents. They also work on both mobile and desktop, so you can visit your favorite site anywhere.

These open web technologies became the default experience for Chrome late last year when sites

https://www.blog.google/products/chrome/saying-goodbye-flash-chrome/

B.3k User After Free

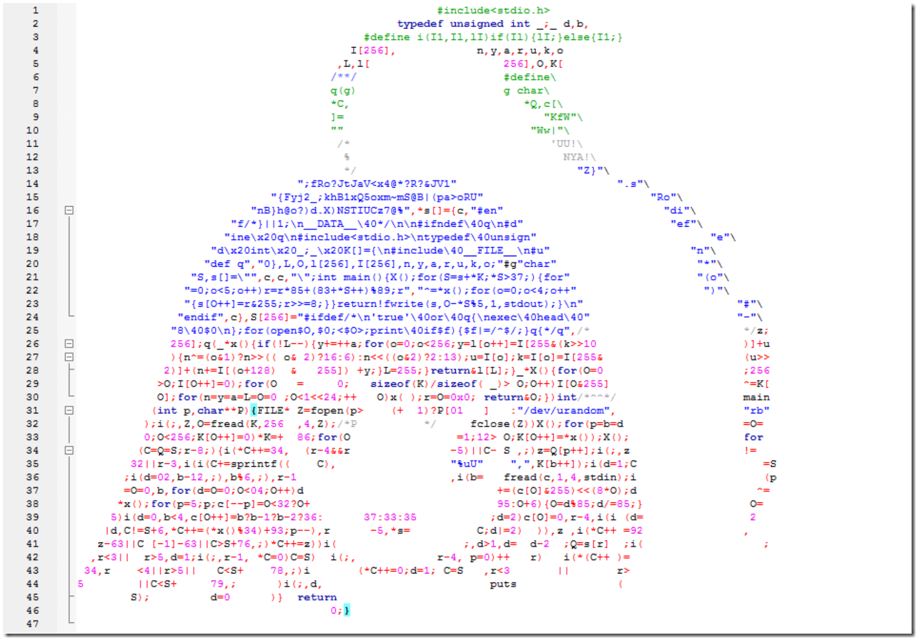
Adobe Flash Player: Exploiting CVE-2015-0311



https://www.coresecurity.com/core-labs/articles/exploiting-cve-2015-0311-a-use-after-free-in-adobe-flash-player

B.3| UAF vs BOF

Trusting the Programmer?



International Obfuscated C Code Contest: https://www.ioccc.org/

- Buffer overflows rely on the ability to read or write outside the bounds of a buffer
- Use-after-free relies on the ability to keep using freed memory once it's been reallocated
- C and C++ programs expect the programmer to ensure this never happens
- But humans (regularly) make mistakes!

B.4 Command Injection

Out of Type Attack

Type safety will not rule out all forms of attack

Command Injection: (also known as shell injection) is a security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application.

B.4b Unsecured Input: Provided data as Code

Attacker tricks an application to treat attacker-provided data as code

This feature appears in many exploits too

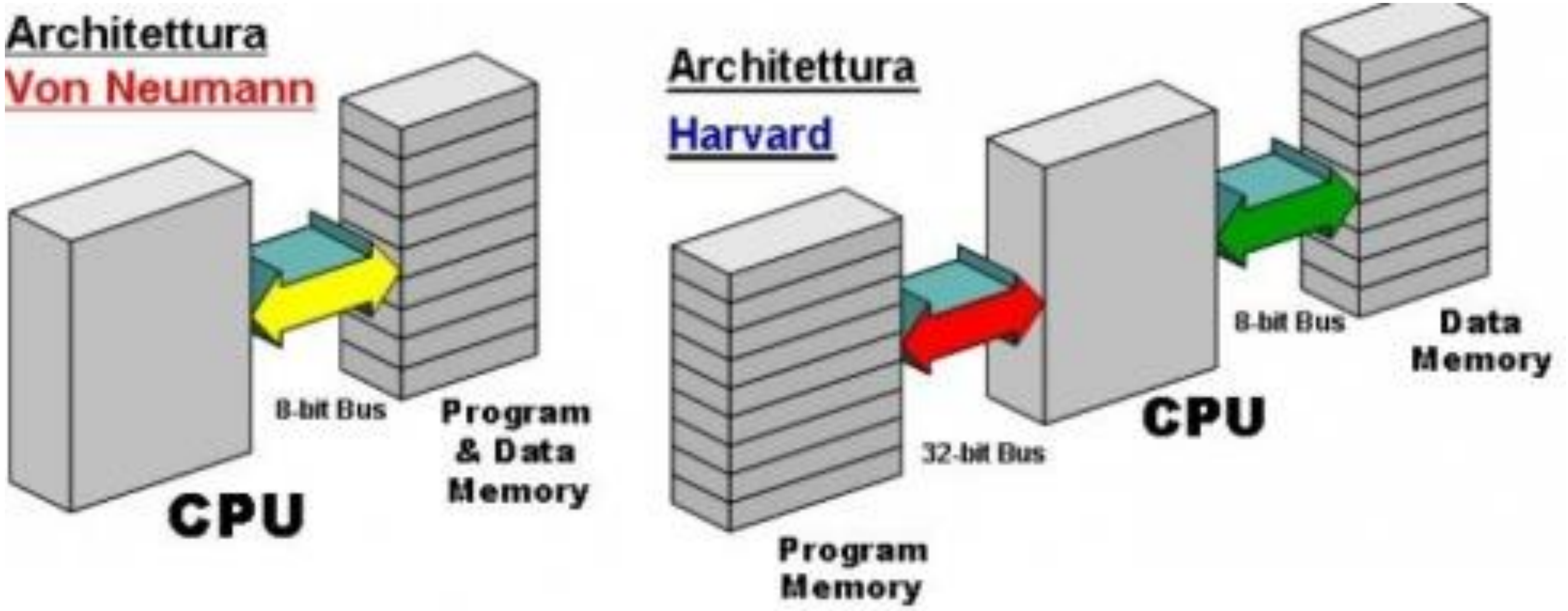
- SQL injection treats data as database queries
- Cross-site scripting treats data as Javascript commands
- Command injection treats data as operating system commands
- Etc.

B.4c Unsecured Input: Provided data as Code

von Neumann vs Harward Architecture

Tricking an application to treat provided data as code

Architettura di von Neumann vs Architettura di Harward



B.4d Unsecured Input: Provided data as Code

Command Injection

What’s wrong in this code?

Ruby code

```
if ARGV.length < 1 then
puts "required argument: textfile path"
exit 1
end
# call cat command on given argument
system(“cat ”+ARGV[0])
exit 0
```

B.4e Unsecured Input: Provided data as Code

Possible Interactions

```
> cat hello.txt
Hello world!
> ls
catwrapper.rb
hello.txt
> ruby catwrapper.rb hello.txt
Hello world!
> ruby catwrapper.rb catwrapper.rb
if ARGV.length < 1 then
puts "required argument: textfile path”
…
> ruby catwrapper.rb “hello.txt; rm hello.txt”
Hello world!
> ls
catwrapper.rb
```

B.4e Unsecured Input: Provided data as Code

What Happened


```
if ARGV.length < 1 then
puts "required argument: textfile path"
exit 1
end
# call cat command on given argument
system(“cat ”+ARGV[0])
exit 0
```

system() interpreted the string as having two commands, and executed them both

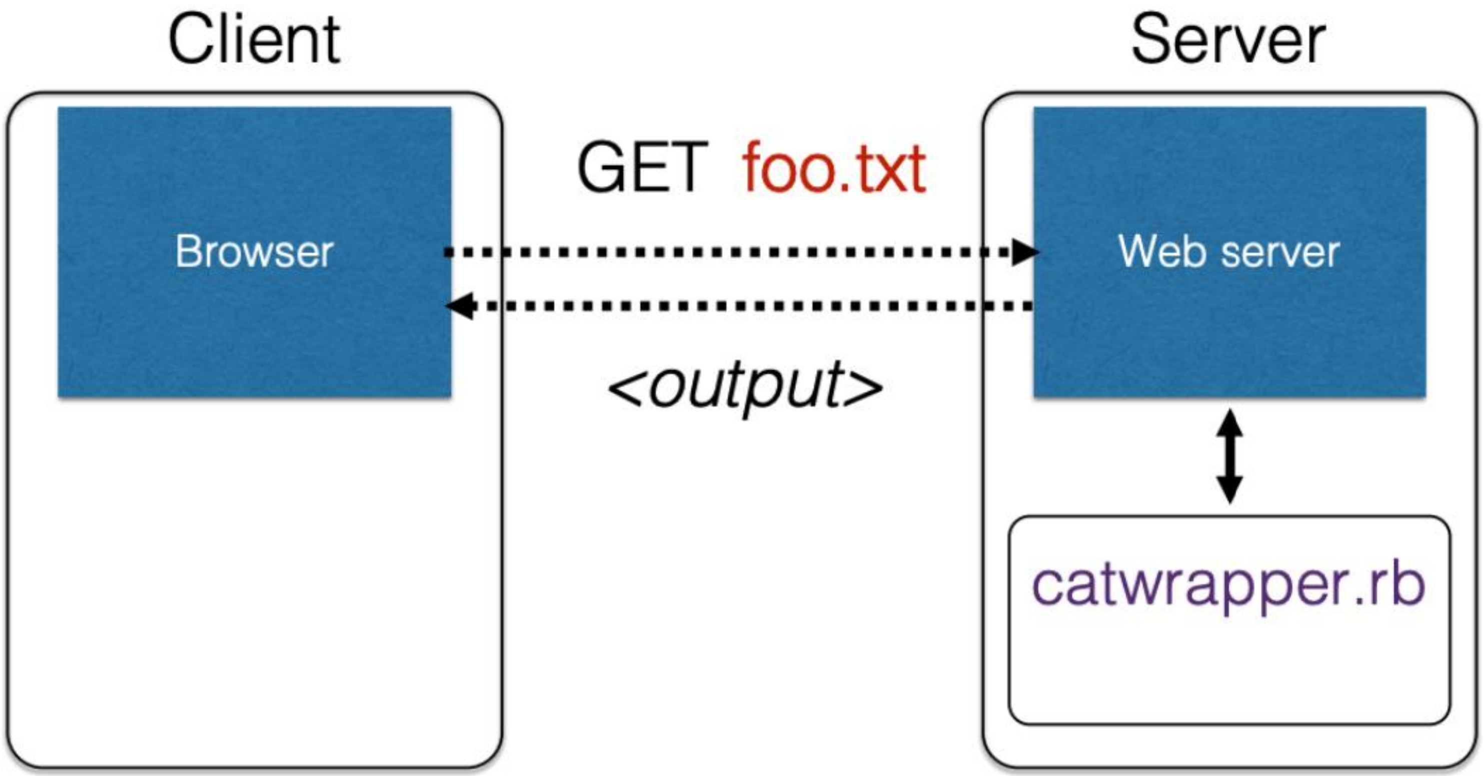
B.4f Unsecured Input: Provided data as Code

If the script were part of a web service…

Input is untrusted — could be anything

But the requestors should only be able to read (see) the contents of the files, not to write (delete)

Current code is too powerful



catwrapper.rb as a web service

The code is vulnerable to **Command Injection**

To fix it **Input Validation**

https://www.owasp.org/index.php/Command_Injection

B.4g Unsecured Input: Provided data as Code

Input Validation

Making input trustworthy

- **Sanitize** it by modifying it or using it it in such a way that the result is correctly formed by construction
- Check it has the expected form, and reject it if not

https://www.owasp.org/index.php/Command_Injection

B.4h Unsecured Input: Defenses

Input treatment Options

break input treatment options down in a number of types:

•**Checking Whitelisting:** reject strings that seems invalid (safer than fix it). ➔ Principle of “Fail Safe” by default

•**Sanitization Escaping:** Replace problematic characters with safe ones.

•**Checking Blacklisting:** Reject strings with possibly bad chars.

•**Sanitization Blacklisting:** Delete the characters you don’t want.

B.4h1 Unsecured Input: Defenses

Checking: Whitelisting

Check the user input to recognize as safe (e.g. proper filename → intensive description)

```
if ARGV.length < 1 then
puts "required argument: textfile path"
exit 1
files = Dir.entries(".").reject{|f| File.directory?(f)}
if not (files.member? ARGV[0]) then
puts "illegal argument"
exit 1
else
# call cat command on given argument
system(“cat ”+ARGV[0])
exit 0
end
```

reject inputs that do not mention a legal file name

B.4h2 Unsecured Input: Defenses

Sanitization: Escaping

Replace problematic chars with safer ones (→ extensive description): ’ → \’ → \; - → \- \ → \\

```
if ARGV.length < 1 then
puts "required argument: textfile path"
exit 1
def escape_chars(string)
pat = /(\'|\"|\.|\*|\/|\-|\\|;|\||\s)/
string.gsub(pat){|match|"\\" + match}
End
# call cat command on given argument
system(“cat ”+ARGV[0])
exit 0
end
```

Escape occurrences of ‘, “”, ; etc. in input string

ARGV “hello.txt; rm hello.txt”
Becomes
“hello.txt rm hello.txt”
→ System response
cat: hello.txt rm hello.txt: No such file or directory

B.4h3 Unsecured Input: Defenses

Checking: Blacklisting

Reject strings with possibly bad chars (→ extensive description): ’ ; --

```
if ARGV.length < 1 then
puts "required argument: textfile path"
exit 1
if ARGV[0] =~ /;/ then
puts "illegal argument"
exit 1
# call cat command on given argument
system(“cat ”+ARGV[0])
exit 0
end
```

Reject input strings with: ;

B.4h4 Unsecured Input: Defenses

Sanitization: Blacklisting

Delete unwanted chars from the input string (→ extensive description): ’ ; --

```
if ARGV.length < 1 then
puts "required argument: textfile
path"
exit 1
# call cat command on given
argument
system(“cat +ARGV[0].tr(“;”,“”))
exit 0
end
```

Delete occurencies of ; from the input string

B.4i Unsecured Input: Defenses

Validation Challenges

Summary of validation actions and their challenges:

Defense	Summary	Description	Challenge
Checking Whitelisting	reject strings that seems invalid (safer than fix it).	Intensive	Cannot always identify whitelist cheaply or completely • May be expensive to compute at runtime • May be hard to describe (e.g., “all possible proper names”)
Sanitization Escaping	Replace problematic characters with safe ones	Extensive	Cannot always delete or sanitize problematic characters • You may want dangerous chars, e.g., “Peter O’Connor” • How do you know if/when the characters are bad? • Hard to think of all of the possible characters to eliminate
Checking Blacklisting	Reject strings with possibly bad chars	Extensive	
Sanitization Blacklisting	Delete the characters you don’t want	Extensive	

B.4j Defenses

Risk treatment Options

break risk treatment options down in a number of types:

•**Avoid:** Risk avoidance is actually pretty self-explanatory. If a risk is deemed too high, then you simply **avoid the activity that creates the risk**. For instance, if flying in an airplane is too risky, you avoid taking the flight in the first place, and completely avoid the risk. Another example would be hiring an individual whose references would not recommend rehiring him — by not hiring him, you avoid the risk that he would not be an asset to your company.

•**Transfer:** In many instances, you can transfer the risk you take to another party. For instance, **insurance** companies exist for exactly this reason. You can also outsource the process in which the risk is present to another provider, thereby transferring the risk to the outsource provider.

•**Reduce:** Risk reduction is one of the **most crucial steps** for processes or activities that **cannot be avoided**, and where risk **cannot be transferred** to another party. An example of this would be training your staff on how to identify a phishing email, or on best practices involving login credentials and password hygiene.

•**Accept:** For some processes and activities, there is no option but to **accept the risk**. Of course, these instances should only involve **low risk**, or repercussions that are easily managed. Some risks might be completely acceptable and require you to take no action at all (a missed deadline on an open-ended project schedule, for instance).

B.4k Defenses

Risk treatment Options

break risk treatment options down in a number of types:

Option			
Avoid	avoid the activity that creates the risk	**Checking Whitelisting**	reject strings that seems invalid (safer than fix it).
Transfer	transfer the risk you take to another party	**Sanitization Escaping**	Replace problematic characters with safe ones
Reduce	security actions for reducing the vulnerabilities	**Checking Blacklisting**	Reject strings with possibly bad chars
Accept	no action at all (or reduced one)	**Sanitization Blacklisting**	Delete the characters you don’t want

B.5 Secure Coding Practice

SEI CERT CODING Standard

The photograph illustrates how the easiest way to break system security is often to circumvent it rather than defeat it (as is the case with most software vulnerabilities related to insecure coding practices).

Top 10 Secure Coding Practices

https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices

B.5b Secure Coding Practice

SEI: Software Engineering Institute

SEI home page

B.5c Secure Coding Practice

SEI CERT CODING Standard

Top 10 Secure Coding Practices

1.Validate input. Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files [Seacord 05].

2.Heed compiler warnings. Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws.

3.Architect and design for security policies. Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.

4.Keep it simple. Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.

5.Default deny. Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted [Saltzer 74, Saltzer 75].

6.Adhere to the principle of least privilege. Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75].

7.Sanitize data sent to other systems. Sanitize all data passed to complex subsystems [C STR02-A] such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.

8.Practice defense in depth. Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05].

9.Use effective quality assurance techniques. Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05].

10.Adopt a secure coding standard. Develop and/or apply a secure coding standard for your target development language and platform.