



IP PARIS



Unsafe System Programming

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2024-09-17



System programming

Programming layered architectures

The architectures of modern computing systems are massively layered. When programming, we target **specific layers**. E.g., from bottom to top:

- (5) (and up...) virtual architectures / virtual machines
- (4) **application level** (business-oriented, frameworks, 4GL)
- (3) **system level** (system languages, system calls, 3GL)
- (2) assembly level (assembly languages, interrupts, 2GL)
- (1) hardware level (firmware, microcode, 1GL)

Each level is characterized by/highly correlated with:

- mechanisms and APIs to interact with lower layers
- suitable **programming languages** (and their generations)

Which layer to target

The choice of layer reveals important trade-offs.

- **Performances.** *Targeting a lower layer might grant better performances.*
 - Writing a performance critical routine in assembly might deliver orders of magnitude speed improvements w.r.t. programming higher layers.
 - “Reimplement at lower layer” is a technique often used for performance-critical code such as device drivers, multimedia codecs, cryptography routines, etc.
- **Portability.** *Targeting a higher layer usually guarantees better portability.*
 - Much better than all lower layer equivalents that might be generated from the chosen layer.
 - E.g.: a block of standard ISO C 99 code can be compiled using `gcc` to more than 70 different target processors.
- **Maintainability.** *Targeting a higher layer usually makes writing code easier and the resulting code more maintainable than if it were written targeting lower layers.*
 - This is largely a consequence of the used programming languages.

System programming — an informal definition

System programming

System programming is the art of writing system software.

— Robert Love, [Linux System Programming: Talking Directly to the Kernel and C Library](#) ↗

System software

System software is “low-level” software that interfaces *directly* with:

- the **kernel** of the operating system
- core **system libraries** (e.g., the C standard library)

Examples

System software that you use daily (possibly without realizing it) includes: **system services** (cron, print spool, power and session management, backup, etc.), **network services** (web servers, mail servers, database management systems, etc.), **toolchains** (shell, compiler, interpreter, debugger).

Try `ps -auxw` on a UNIX shell. *Most* of what you see there is system-level software.

System programming — why bother?

1. **Legacy code**—such as system utilities—is not going away any time soon; in some cases it is also basis for standardization (e.g., UNIX utilities)
 - In the Free Software world, the majority of existing code (50%+ of Debian, see [Barahona et al., 2009](#)) is system-level C code
2. **New system-level tasks** born on a regular basis, to cope with application-level evolution
 - E.g., an increasing number of new applications is written in JavaScript, for the Web, desktops, and mobiles
 - *therefore* we need new and better JavaScript (JIT) compilers; most of their code is system-level code

This course

Observation: *system programs and system programming are here to stay.*

This course main topic: **how to make system programming safe.**

Unsafe system programming

“Convert a String to Uppercase in C”

Quasi-verbatim quote from [Tutorials Point solution](#) to this problem ([archived copy](#)).
(Because you never blindly reuse code snippets from the Web or ChatGPT, do you?)

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char s[100];
6     int i;
7     printf("\nEnter a string : ");
8     gets(s);
9     for (i = 0; s[i] != '\0'; i++) {
10         if(s[i] >= 'a' && s[i] <= 'z') {
11             s[i] = s[i] - 32;
12         }
13     }
14     printf("\nString in Upper Case = %s\n", s);
15     return 0;
16 }
```

Q: what's the problem with this code?

man 3 gets (before ISO C11)

gets - get a string from standard input

SYNOPSIS

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

```
char *gets(char *s);
```

DESCRIPTION

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

RETURN VALUE

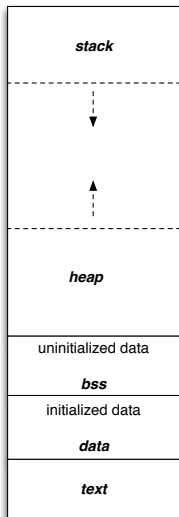
gets() returns s on success, and NULL on error or when end of file occurs while no characters have been read. However, given the lack of buffer overrun checking, there can be no guarantees that the function will even return.

BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security.

Memory layout — redux

The memory layout of a running program is organized in **memory segments**.



BSS

- Historical acronym for “Block Starting Symbol”
- **Uninitialized static data**
- E.g., global variables and local static variables with no initial value

Data

- **Initialized static data**
- E.g., global variables and local static variables *with* initial values

Text (Code)

- **Executable code**
- In a memory segment marked as: executable and (usually) read-only

Memory layout — redux (cont.)

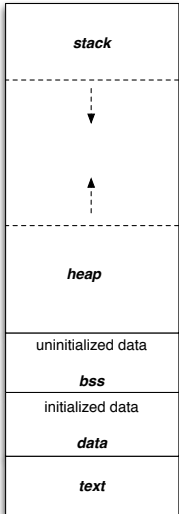
The memory layout of a running program is organized in **memory segments**.

Stack

- Call stack: one **stack frame** per function call in execution
- LIFO (last-in, first-out) order, growing downward (toward lower memory addresses)
- **Stack pointer** (SP) registry pointing to the top of the stack (lowest address)
- Each stack frame contains: **return address** of the function being executed, **automatic variables** (local function variables and arguments)

Heap

- **Dynamically allocated memory** (e.g., via `malloc`)
- Growing upward (toward higher memory addresses)



Memory layout — example

```
1  /* Definition of an initialized global variable */
2  int x_global = 1;
3
4  /* Declaration of a global variable that exists somewhere else */
5  extern int y_global;
6
7  /* Declaration of a function that exists somewhere else */
8  int fn_a(int, int, int, int);
9
10 /* Definition of a function. */
11 int fn_b(int x_local) {
12     /* Definition of an initialized local variable */
13     int y_local = 3;
14     static int z_static = 4;
15
16     /* Code that refers to local and global variables and other functions. */
17     x_global = fn_a(x_local, x_global, y_local, y_global);
18     z_static += 1;
19     return (x_global + z_static);
20 }
21 /* end of anatomy.c */
```

Memory layout — example (cont.)

Non-dynamic segments of the memory layout of executable programs have counterparts that can be observed in compiled objects using the `nm` UNIX tool.

```
$ gcc -Wall -c anatomy.c
$ nm anatomy.o
                 U fn_a
0000000000000000 T fn_b
0000000000000000 D x_global
                 U y_global
0000000000000004 d z_static.0
```

- U** undefined symbol (code or data segment, not present in this `.o` linking object)
- T** defined text symbol (code segment, present in this object)
- D/d** defined data symbol (data segment, present in this object). Lowercase if local (e.g., not exported), uppercase if global

Buffer overflows

Let's go back to our `gets()` example:

“No check for buffer overrun is performed.” → What can possibly go wrong?

- A **data buffer** is a memory region used to store data in transit from one place to another.
- At any given point in time a buffer has a fixed size n , usually measured in bytes; buffer boundaries correspond to those of the integer interval $[0, n-1]$.
- In (many, most) system programming languages there is no runtime check to prevent **writing beyond** buffer boundaries.

Buffer overflow

A **buffer overflow** is a *program anomaly* that occurs when a data write targeting a buffer writes outside the buffer boundaries, overwriting neighboring memory.

How severe a buffer overflow is depends on what happens to reside in “neighboring memory”...

Buffer overflow — example

```
1 #include <string.h>
2
3 void foo(char *bar) {
4     char c[12];
5     strcpy(c, bar); // no bounds checking
6 }
7
8 int main(int argc, char **argv) {
9     foo(argv[1]);
10    return 0;
11 }
```

SYNOPSIS

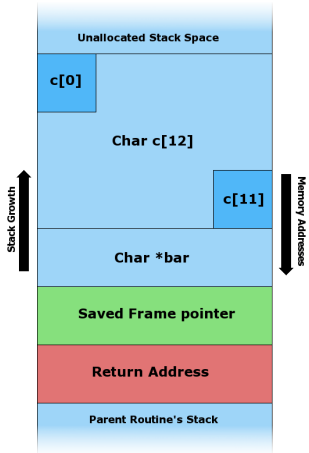
```
char *strcpy(char *restrict dest, const char *restrict src);
```

DESCRIPTION

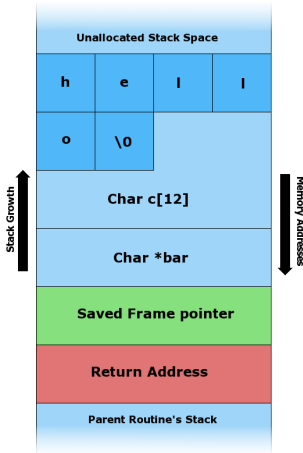
The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy. Beware of buffer overruns! (See `BUGS`.)

(Sounds familiar?)

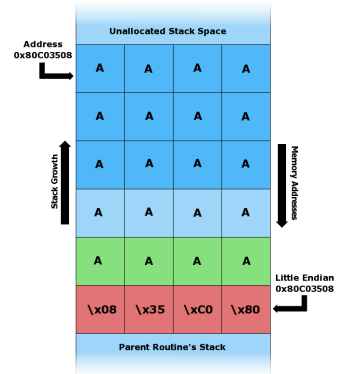
Buffer overflow — example (cont.)



Before `strcpy`



After `strcpy`, when
`argv[1] = "hello"`



After `strcpy`, when
`argv[1] =`
`"AA[...]AA\x08\x3"`

Buffer overflows (cont.)

From there on, all bets are off.

- The attacker controls the return point of the function in execution.
- They can make it point to an existing different function (altering the execution flow, e.g., to bypass the execution of security checks).
- They can make it point to code that is *itself* passed as an argument (**arbitrary code execution**), provided that they can write to an executable memory segment.
- The attacker can also **overwrite** the content of **local variables** of other functions, not only the return address, e.g., to inject authentication credentials.

Good reads

- *Aleph One*. [Smashing the stack for fun and profit](#). Phrack magazine 7.49 (1996): 14-16.
- *Jon Gjengset*. [Smashing the Stack in the 21st Century](#). Blog post, 2019.

`gets()` (as many other functions...) from our previous example can fail in exactly the same way as `strcpy()` here, except the input would come from `stdin` instead of `argv`.

man 3 gets (ISO C11)

And indeed the doc of *modern gets* says:

```
gets - get a string from standard input (DEPRECATED)
```

SYNOPSIS

```
[[deprecated]] char *gets(char *s);
```

DESCRIPTION

Never use this function.

`gets()` reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or EOF, which it replaces with a null byte (`'\0'`). No check for buffer overrun is performed (see **BUGS** below).

BUGS

```
[..] Use fgets() instead.
```

The `[[deprecated]]` attribute makes compilers trigger warnings at compile time (by default).

YAY, problem solved! **Good developers** these days would never do this, right? ... **RIGHT?**

Cars Attacks!

Checkoway et al. [Comprehensive Experimental Analyses of Automotive Attack Surfaces](#). USENIX Security Symposium 2011.

- *“Like many modern cars, our car’s cellular capabilities facilitate a variety of safety and convenience features (e.g. the car can automatically call for help if it detects a crash). However, long-range communication channels also offer an obvious target for potential attackers...”*
- *“To synthesize a digital channel in this environment, the manufacturer uses Airbiquity’s aqLink software modem [analog radio, as backup for missing cellular network at the time] to convert between analog waveforms and digital bits.”*
- *“the aqLink code explicitly supports packet sizes up to 1024 bytes. However, the custom code that glues aqLink to the Command program assumes that packets will never exceed 100 bytes or so (presumably since well-formatted command messages are always smaller). This leads to another stack-based buffer overflow vulnerability that we verified is exploitable.”*

Cars Attacks! (cont.)

Checkoway et al. [Comprehensive Experimental Analyses of Automotive Attack Surfaces](#). USENIX Security Symposium 2011.

- *“We also found that the entire attack can be implemented in a completely blind fashion— without any capacity to listen to the car’s responses. Demonstrating this, we encoded an audio file with the modulated post-authentication exploit payload and loaded that file onto an iPod. By manually dialing our car on an office phone and then playing this “song” into the phone’s microphone, we are able to achieve the same results and compromise the car”*

Discuss

10+ years later, do you think the situation of cars security is any better or any worse? Why?

Some dangerous software weaknesses

Background — Terminology about known vulnerabilities: CVE, CWE

Common Vulnerabilities and Exposures (CVE)

*The Common Vulnerabilities and Exposures (CVE) system provides a **reference-method** for publicly known information-security vulnerabilities and exposures. The United States' National Cybersecurity FFRDC, operated by The MITRE Corporation, maintains the system [...]. The system was officially launched for the public in September 1999. (Wikipedia ↗)*

Examples of public vulnerability databases using CVEs: <https://cve.mitre.org>, <https://nvd.nist.gov>, <https://osv.dev>.

Common Weakness Enumeration (CWE)

*The Common Weakness Enumeration (CWE) is a **category system** for hardware and software weaknesses and vulnerabilities. It is sustained by a community project with the goals of understanding flaws in software and hardware and creating automated tools that can be used to identify, fix, and prevent those flaws. The project is sponsored by the National Cybersecurity FFRDC, which is operated by The MITRE Corporation [...]. (Wikipedia ↗)*

Moar buffer overflows in vulnerability databases

Search results for “buffer overflow” at https://cve.mitre.org/cve/search_cve_list.html (2022-12-09):

Search Results

There are **13521** CVE Records that match your search.

Name	Description
CVE-2022-46824	In JetBrains IntelliJ IDEA before 2022.2.4 a buffer overflow in the fsnotifier daemon on macOS was possible.
CVE-2022-45672	Tenda i22 V1.0.0.3(4687) was discovered to contain a buffer overflow via the formWx3AuthorizeSet function.
CVE-2022-45671	Tenda i22 V1.0.0.3(4687) was discovered to contain a buffer overflow via the appData parameter in the formSetAppFilterRule function.
CVE-2022-45670	Tenda i22 V1.0.0.3(4687) was discovered to contain a buffer overflow via the ping1 parameter in the formSetAutoPing function.
CVE-2022-45669	Tenda i22 V1.0.0.3(4687) was discovered to contain a buffer overflow via the index parameter in the formWifiMacFilterGet function.
CVE-2022-45664	Tenda i22 V1.0.0.3(4687) was discovered to contain a buffer overflow via the list parameter in the formWrlSSIDget function.
CVE-2022-45663	Tenda i22 V1.0.0.3(4687) was discovered to contain a buffer overflow via the index parameter in the formWifiMacFilterSet function.
CVE-2022-45661	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the time parameter in the setSmartPowerManagement function.
CVE-2022-45660	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the schedStartTime parameter in the setSchedWifi function.
CVE-2022-45659	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the wpapsk_crypto parameter in the fromSetWirelessRepeat function.
CVE-2022-45658	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the schedEndTime parameter in the setSchedWifi function.
CVE-2022-45657	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the list parameter in the fromSetIpMacBind function.
CVE-2022-45656	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the time parameter in the fromSetSysTime function.
CVE-2022-45655	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the timeZone parameter in the form_fast_setting_wifi_set function.
CVE-2022-45654	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the ssid parameter in the form_fast_setting_wifi_set function.
CVE-2022-45653	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the page parameter in the fromNatStaticSetting function.
CVE-2022-45652	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the startIp parameter in the formSetPPTPServer function.
CVE-2022-45651	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the list parameter in the formSetVirtualSer function.
CVE-2022-45650	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the firewallEn parameter in the formSetFirewallCfg function.
CVE-2022-45649	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the endIp parameter in the formSetPPTPServer function.
CVE-2022-45648	Tenda AC6V1.0 V15.03.05.19 was discovered to contain a buffer overflow via the devName parameter in the formSetDeviceName function.

At the time the database contained 190 175 CVE records, 7% of which were related to buffer overflows.

How about other memory-related issues?

2022 CWE Top 25 Most Dangerous Software Weaknesses

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

Source: [MITRE](#) .

The KEV Count shows the number of 2020 and 2021 CVEs that are **known to have been exploited** in the wild (hence there are *more* CVEs related to these CWEs, either not exploited or not known to have been).

Can you spot the memory-related vulnerability classes?

2022 CWE Top 25 vs memory safety

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

Out-of-bounds write (CWE-787)

Description

The software writes data past the end, or before the beginning, of the intended buffer.

- Impact: data corruption, crash, code execution
- We have already seen two examples of this
- There are more! <https://cwe.mitre.org/data/definitions/787.html>

Let's move on to the other classes.

Out-of-bounds read (CWE-125) — example

Description

The software reads data past the end, or before the beginning, of the intended buffer.

- Impact: read sensitive information, crash (e.g., due to segmentation fault)
- Spot the bug (level: easy):

```
1 int getValueFromArray(int *array, int len, int index) {
2     int value;
3     if (index < len) {
4         value = array[index];
5     } else {
6         value = 0;
7         errno = 42; // handle error
8     }
9     return value;
10 }
```

- Good: bound check on the right end of the array
- Bad: no bound check on the left end (and `index` can be negative!)

Out-of-bounds read (CWE-125) — example (cont.)

Description

The software reads data past the end, or before the beginning, of the intended buffer.

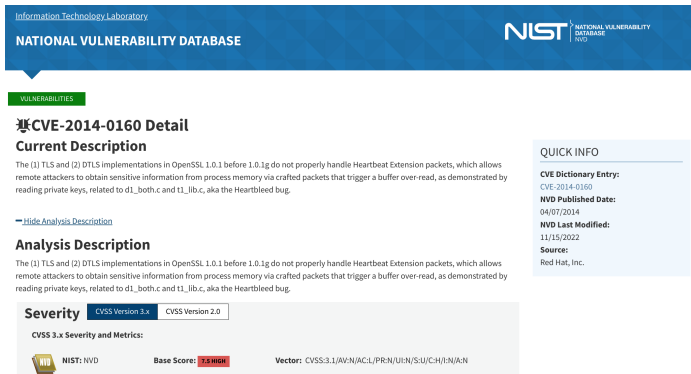
- Spot the bug (level: hard):

```
1 char buffer[128];
2 int bytesToCopy = packet.length;
3 if (bytesToCopy < 128) {
4     strncpy(buffer, packet.data, bytesToCopy);
5 }
```

- Good: bounds check, safe version of `strcpy`
- Bad: cast of **signed** `int bytesToCopy` to **unsigned** `size_t` as 3rd argument of `strncpy`
- More real-world examples: <https://cwe.mitre.org/data/definitions/125.html>

Improper restriction of operations within a buffer (CWE-119) — example

- More general case of “Out-of-bounds Read” (CWE is in fact a *hierarchical taxonomy*)
- With a very (in)famous example: CVE-2014-0160, AKA **Heartbleed**



Information Technology Laboratory
NATIONAL VULNERABILITY DATABASE

NIST NATIONAL VULNERABILITY DATABASE NVD

VULNERABILITIES

CVE-2014-0160 Detail

Current Description

The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1.g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.


[Hide Analysis Description](#)

Analysis Description

The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1.g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.

Severity CVSS Version 3.x CVSS Version 2.0

CVSS 3.x Severity and Metrics:

 NIST: NVD **Base Score: 7.5 HIGH** **Vector:** CVSS:3.1/(AV:N)/(AC:L)/(PR:N)/(UI:N)/(S:U)/(C:H)/(I:N)/(A:N)

QUICK INFO


CVE Dictionary Entry:
CVE-2014-0160

NVD Published Date:
04/07/2014

NVD Last Modified:
11/15/2022

Source:
Red Hat, Inc.



- Fix: [OpenSSL commit 96db9023b881d7cd9f379b0c154650d6c108e9a3](#) 
- More real-world examples: <https://cwe.mitre.org/data/definitions/119.html>

Use After Free (CWE-416) — example

A pointer is used after the memory region it points to is **freed**.

- Spot the bug (level: easy):

```
1 char* ptr = (char*) malloc(SIZE);
2 if (err) {
3     abrt = 1;
4     free(ptr);
5 }
6 /* ... */
7 if (abrt) { log_err("operation aborted before commit", ptr); }
```

- Common causes:
 - Complicated logic to handle errors or other exceptional conditions
 - Confusion about **who is responsible** for deallocating memory
- Impact: reading/writing unexpected memory, segmentation fault, code execution
- More real-world examples: <https://cwe.mitre.org/data/definitions/416.html>

NULL Pointer Dereference (CWE-476) — example

A **NULL** pointer is dereferenced, as if it were pointing somewhere.

- Spot the bug (level: medium):

```
1 void host_lookup(char *user_supplied_addr) {
2     struct hostent *hp;
3     in_addr_t *addr;
4     char hostname[64];
5     in_addr_t inet_addr(const char *cp);
6
7     validate_addr_form(user_supplied_addr);
8     addr = inet_addr(user_supplied_addr);
9     hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);
10    strcpy(hostname, hp->h_name);
11 }
```

- Common causes: lack of pointer verification, multi-thread race conditions
- Impact: crash (which *has* a security impact: why?)
- More real-world examples: <https://cwe.mitre.org/data/definitions/476.html>

Concurrent Execution using Shared Resource with Improper Synchronization (CWE-362, AKA Race Condition) — example

Definition (Race Condition)

A **race condition** is a situation where multiple execution units access shared data concurrently and *the outcome of the execution depends on the particular order* in which accesses take place.

Not all race conditions are *security* issues. A race is a security issue when one or more of the possible outcomes violates security requirements.

- Spot the bug (level: easy):

```
1  $transfer_amount = GetTransferAmount();
2  $balance = GetBalanceFromDatabase();
3  if ($transfer_amount < 0) {
4      FatalError("Bad Transfer Amount");
5  }
6  $newbalance = $balance - $transfer_amount;
7  if (($balance - $transfer_amount) < 0) {
8      FatalError("Insufficient Funds");
9  }
10 SendNewBalanceToDatabase($newbalance);
11 NotifyUser("Transfer of $transfer_amount succeeded. New balance: $newbalance.");
```


Race Condition (CWE-362) — example (cont.)

Mutexes (mutual-exclusion locks) are among the common programming abstractions available to developers to avoid race conditions.

Unfortunately, they are also notoriously hard to use correctly...

- Spot the bug (level: medium):

```
1 void f(pthread_mutex_t *mutex) {  
2     pthread_mutex_lock(mutex);  
3  
4     /* access shared resource */  
5  
6     pthread_mutex_unlock(mutex);  
7 }
```

How do we fix this?

Whose fault is this?

Discuss

- Don't blame the developers!
 - ... if you give them **bad tools** for the job
 - ... if they work in **bad conditions** (e.g., crunches), even “only” sometimes
 - ... if they are **not the right persons** for the job (the developer job market is crazy!)
- Do you think *you* will *always* be able to avoid introducing this kind of bugs?

This course

State-of-the art solutions for either minimizing the risk of or avoiding entirely (by construction) the introduction of bugs in system programs that could have a severe security impact.

Credits

- These slides contain material and ideas reused with permission from lecture 1 of Stanford's course [CS 110L](#) (2022) by Ryan Eberhardt, Armin Namavari, Will Crichton, Julio Ballista, and Thea Rossman.
- Images:
 - Memory layout image from [Wikipedia](#), by Dougct, license [CC BY-SA 3.0](#).
 - Stack overflow images from [Wikipedia](#), by [Michael Lynn](#), public domain.