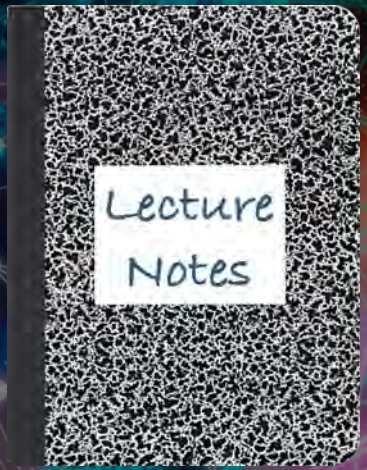
CS 419: Computer Security

Week 7: Memory Corruption & Code Injection



Paul Krzyzanowski

Part 1

Hijacking & Injection

Two New Windows Zero-Days Exploited in the Wild — One Affects Every Version Ever Shipped

October 15, 2025 • Ravie Lakshmanan

Microsoft on Tuesday released fixes for a whopping 183 security flaws spanning its products, including three vulnerabilities that have come under active exploitation in the wild, as the tech giant officially ended support for its Windows 10 operating system unless the PCs are enrolled in the Extended Security Updates (ESU) program.

Of the 183 vulnerabilities, eight of them are non-Microsoft issued CVEs. As many as 165 flaws have been rated as Important in severity, followed by 17 as Critical and one as Moderate. The vast majority of them relate to elevation of privilege vulnerabilities (84), with remote code execution (33), information disclosure (28), spoofing (14), denial-of-service (11), and security feature bypass (11) issues accounting for the rest of them.

The updates are in addition to the 25 vulnerabilities Microsoft addressed in its Chromium-based Edge browser since the release of September 2025's Patch Tuesday update.

https://thehackernews.com/2025/10/two-new-windows-zero-days-exploited-in.html

Two New Windows Zero-Days Exploited in the Wild — One Affects Every Version Ever Shipped

October 1

Microsoft
vulnerabili
its Windov

Of the 183
as Importa
elevation
(14), denia

The updat
since the r

CVE-2025-24990 (CVSS score: 7.8)

Windows Agere Modem Driver ("ltmdm64.sys") Elevation of Privilege Vulnerability

Weakness: Untrusted Pointer Dereference

What privileges could be gained by an attacker who successfully exploited this vulnerability?

An attacker who successfully exploited this vulnerability could gain administrator privileges.

Is the vulnerability only exploitable if the Agere Modem is actively being used?

No. All supported versions of Windows can be affected by a successful exploitation of this vulnerability, even if the modem is not actively being used..

Two New Windows Zero-Days Exploited in the Wild — One Affects Every Version Ever Shipped

October 1

Microsoft
vulnerabili
its Windov

Of the 183
as Importa
elevation
(14), denia

The updat
since the r

CVE-2025-49708 (CVSS score: 9.9)

Microsoft Graphics Component Elevation of Privilege Vulnerability

Weakness: Use After Free

What privileges could be gained by an attacker who successfully exploited this vulnerability?

An attacker who successfully exploited this vulnerability could gain SYSTEM privileges.

How could an attacker exploit this vulnerability?

An attacker can exploit this vulnerability by getting access to the local guest VM so they can attack the Host OS.

According to the CVSS metric, a successful exploitation could lead to a scope change (S:C).

What does this mean for this vulnerability?

Compromising the host enables an attacker to impact other virtual machines running on the same host, even if those VMs are not directly vulnerable to this issue.

Two New Windows Zero-Days Exploited in the Wild — One Affects Every Version Ever Shipped

October 1

Microsoft
vulnerabili
its Windov

Of the 183
as Importa
elevation
(14), denia

The updat
since the r

CVE-2025-2884 (CVSS score: 4.6)

Trusted Computing Group (TCG) TPM2.0 reference implementation's CryptHmacSign helper function

Weakness: Out of Bounds Read

Not yet exploited

Two New Windows Zero-Days Exploited in the Wild — One Affects Every Version Ever Shipped

October 1

Microsoft
vulnerabili
its Windov

Of the 183
as Importa
elevation
(14), denia

The updat
since the r

CVE-2025-59287 (CVSS score: 9.8)

Windows Server Update Service (WSUS) Remote Code Execution Vulnerability

Weakness: Deserialization of Untrusted Data

How could an attacker exploit the vulnerability?

A remote, unauthenticated attacker could send a crafted event that triggers unsafe object deserialization in a legacy serialization mechanism, resulting in remote code execution.

Hijacking & Injection

Hijacking: Taking control of a process by intercepting, manipulating, or redirecting its intended behavior for unintended purposes without injecting new code

- Session hijacking: take over someone's authenticated session
  - Snoop on a communication session to get authentication info
  - Access someone's cookies for a web session
  - Perform an Adversary-in-the-Middle (AitM) attack to let a user log in and use that session
- Control flow hijacking: alter program execution
  - Use *return-to-libc* or *return-oriented programming* techniques to alter execution
- Other forms of hijacking
  - Browser redirection hijacking: Redirect a victim's web browser to a malicious site
  - Domain hijacking: Change DNS (IP address lookup) results to direct users to malicious addresses
  - Search Engine Poisoning: Change the browser's default search engine

Hijacking & Injection

Injection

Inserting arbitrary code or commands into a process to execute unintended operations

- Command injection: get a process to run arbitrary system commands
  - Send commands to a program that are then executed by the system shell
  - Includes SQL injection – send database commands
- Code injection: get a process to run arbitrary code
  - Overflow an input buffer and cause new code to run
  - Provide JavaScript as input that will later get executed (Cross-site scripting)
- Library injection: have a process run with different linked libraries
  - Alter the search path or force a program to load alternate DLL/shared libraries

Security-Sensitive Programs & Remote Services

Hijacking or injection isn't interesting for regular programs on your system

- You might as well just run the commands from the shell or write a program

- It *is* interesting if
  - The program runs with elevated privileges (*setuid*), especially if it runs as root
  - Runs on a system you don't have access to (most servers)
    - This is Remote Code Execution (RCE)

- It is super interesting if
  - The program runs with elevated privileges on a **remote system** you can't access directly

Bugs and mistakes

- Most attacks are due to
  - Social engineering: getting a legitimate user to do something
  - Or exploiting vulnerabilities: using a program in a way it was not intended
    - This includes buggy security policies
- An attacked system may be further weakened because of poor access control rules
  - Allowing the attacker to do more than the compromised application – a violation of the *Principle of Least Privilege*
- Cryptography won't save us!
  - And cryptographic software can also be buggy

Unchecked Assumptions

- Unchecked assumptions can lead to vulnerabilities
  - Vulnerability: weakness that can be exploited to perform unauthorized actions
- Attack
  - Discover these assumptions
  - Craft an **exploit** to render them invalid … and run the exploit
- Four common assumptions:
  1. The buffer is large enough for the data
  2. Integer overflow doesn't exist
  3. User input will never be processed as a command
  4. A file is in a proper format

Memory Corruption Vulnerabilities

- Stack attacks
  - Buffer overflow: writing more data that a buffer can hold overwrites adjacent memory
- Heap attacks
  - Exploit vulnerabilities in dynamic memory allocation
  - Heap overflow: write beyond allocated space (a buffer overflow)
  - Use-After-Free: access memory after it's been freed (and possibly reallocated)
- Integer overflow/underflow
  - Arithmetic operation exceeds the maximum or minimum value a data type can hold
  - This can lead to unexpected behavior like buffer overflows or bad logic

Goal: How memory errors lead to code execution

Stack Buffer Overflow

What is a buffer overflow?

Programming error that allows more data to be stored in an array than there is allocated space for the object

- Buffer = chunk of memory on the stack, heap, or static data
- Overflow means adjacent memory will be overwritten
  - Program data can be modified
  - New code can be injected
  - Unexpected transfers of control can be launched

Buffer overflows

Buffer overflows used to be responsible for ~50% of vulnerabilities

- We know how to defend ourselves but
  - Average time to discover and patch a bug is more than 1 year
  - People delay updating systems … or refuse to
  - Embedded systems often never get patched
    - Routers, cable modems, set-top boxes, access points, IP phones, and security cameras
  - Embedded systems often don't defend against this (in the name of efficiency)
  - Insecure access rights often help with gaining access or more privileges
  - We continue to write buggy code!

cve.mitre.org reports 4,353 CVE records for buffer overflows in 2025 so far
2,103 vulnerabilities in 2024

The classic buffer overflow bug

gets.c from macOS:
© 1990,1992 The Regents of the University of California.

```
gets(buf)
char *buf;
  register char *s;
  static int warned;
  static char w[] = "warning: this program uses gets(),
  which is unsafe.\r\n";

  if (!warned) {
      (void) write(STDERR_FILENO, w, sizeof(w) - 1);
      warned = 1;
  }
  for (s = buf; (c = getchar()) != '\n';)
      if (c == EOF)
            if (s == buf)
                  return (NULL);
            else
                  break;
      else
            *s++ = c;
  *s = 0;
  return (buf);
}
```

gets.c from OS X: © 1990,1992 The Regents of the University of California.

```
gets(buf)
char *buf;
  register char *s;
  static int warned;
```

```
for (s = buf; (c = getchar()) != '\n';)
  if (c == EOF)
     if (s == buf)
        return (NULL);
     else
        break;
  else
     *s++ = c;
```

```
  return (buf);
}
```

gets(),

1);

Note there's no check for the length of the buffer!

An issue with C++ too – and no warnings!

```
#include <iostream>

using namespace std;

int main()
{
     char x[4] = "cat";
     char y[4];
     char z[4] = "dog";

     cout << "Enter a word:";
     cin >> y;
     cout << "Read " << strlen(y) << " characters." << endl;
     cout << "x: " << x << endl;
     cout << "y: " << y << endl;
     cout << "z: " << z << endl;
}
```

An issue with C++ too – and no warnings!

```
#include <iostream>

using names

int main()
{
     char x[
     char y[
     char z[

     cout <<
     cin >>
     cout <<
     cout <<
     cout <<
     cout <<
}
```

```
$ g++ -o cin cin.cpp
Enter a word:abcdefg
Read 7 characters.
x: efg
y: abcdefg
z: dog
```

The data in `y` overflowed to `x`
`x` got corrupted

An issue with C++ too – and no warnings!

```
#include <iostream>

using names

int main()
{
     char x[
     char y[
     char z[

     cout <<
     cin >>
     cout <<
     cout <<
     cout <<
     cout <<
}
```

```
$ g++ -o cin cin.cpp
Enter a word:abcdefghijklmnopqrstuvwxyz0123456789
Read 36 characters.
x: efghijklmnopqrstuvwxyz0123456789
y: abcdefghijklmnopqrstuvwxyz0123456789
z: dog
Bus error: 10
```

With even more data,
`x` got corrupted
AND the program crashed!

Buffer overflow examples

```
void test(void) {
    char name[10];

    strcpy(name, "krzyzanowski");
}
```

That's easy to spot!

Another example

How about this?

```
char configfile[256];
char *base = getenv("BASEDIR");

if (base != NULL)
   sprintf(configfile, "%s/config.txt", base);
else {
   fprintf(stderr, "BASEDIR not set\n");
}
```

Buffer overflow attacks

To exploit a buffer overflow, identify if there's an overflow vulnerability in a program

- Black box testing
  - Trial and error
  - Fuzzing tools (more on that …)
- Inspection
  - Study the source
  - Trace program execution

You don't have access to the source

You have access to the source

Understand where the buffer is in memory and whether there is potential for corrupting surrounding data

What's the harm?

Execute arbitrary code, such as starting a shell

Code injection, *stack smashing*

- Code runs with the privileges of the program
  - If the program is *setuid root* then you have root privileges
  - If the program is on a server, you can run code on that server

- Even if you cannot inject code…
  - You may crash the program (Denial of Service attack)
  - Change how it behaves
  - Modify data

- Sometimes the crashed code can leave a core dump
  - You can access that and grab data the program had in memory

Shellcode

Shellcode:
A compact sequence of machine instructions used as a payload in exploits.

- It is placed into writable memory, and executed by hijacking control flow.
- Commonly spawns a command shell or loads a second-stage payload
- Generally kept small to avoid null bytes and fit in the buffer

Taking advantage of unchecked bounds

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

int
main(int argc, char **argv)
{
      char pass[5];
      int correct = 0;

      printf("enter password: ");
      gets(pass);
      if (strcmp(pass, "test") == 0) {
            printf("password is correct\n");
            correct = 1;
      }
      if (correct) {
            printf("authorized: running with root privileges...\n");
            exit(0);
      }
      else
            printf("sorry - exiting\n");
      exit(1);
}
```

```
$ ./buf
enter password: abcdefghijklmnop
authorized: running with root privileges...
```

Run on my Raspberry Pi 5
Debian 1:6.6.74-1+rpt1
6.6.74+rpt-rpi-2712

Note: this test did *not* succeed

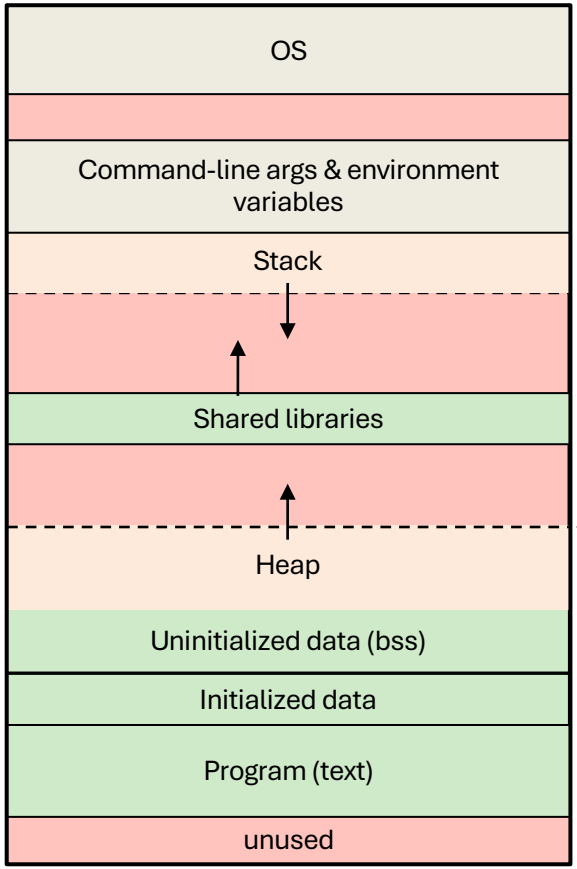
It’s a bounds checking problem

- C and C++
  - Allow direct access to memory
  - Do not check array bounds
  - Functions often do not even know array bounds
    - They just get passed a pointer to the start of an array
- This is not a problem with strongly typed languages
  - Java, C#, Python, etc. check sizes of structures
- But C is in the top 4-5 of popular programming languages
  - #1 choice for system programming & embedded systems
  - Most operating syatems, compilers, interpreters, databases, security appliances, browsers, and libraries are written in C or C++

Part 2

Anatomy of overflows

Linux process memory map*



High memory

0xc0000000

Top of stack (it grows down)

0x40000000

brk

0x08048000

System interrupt vectors

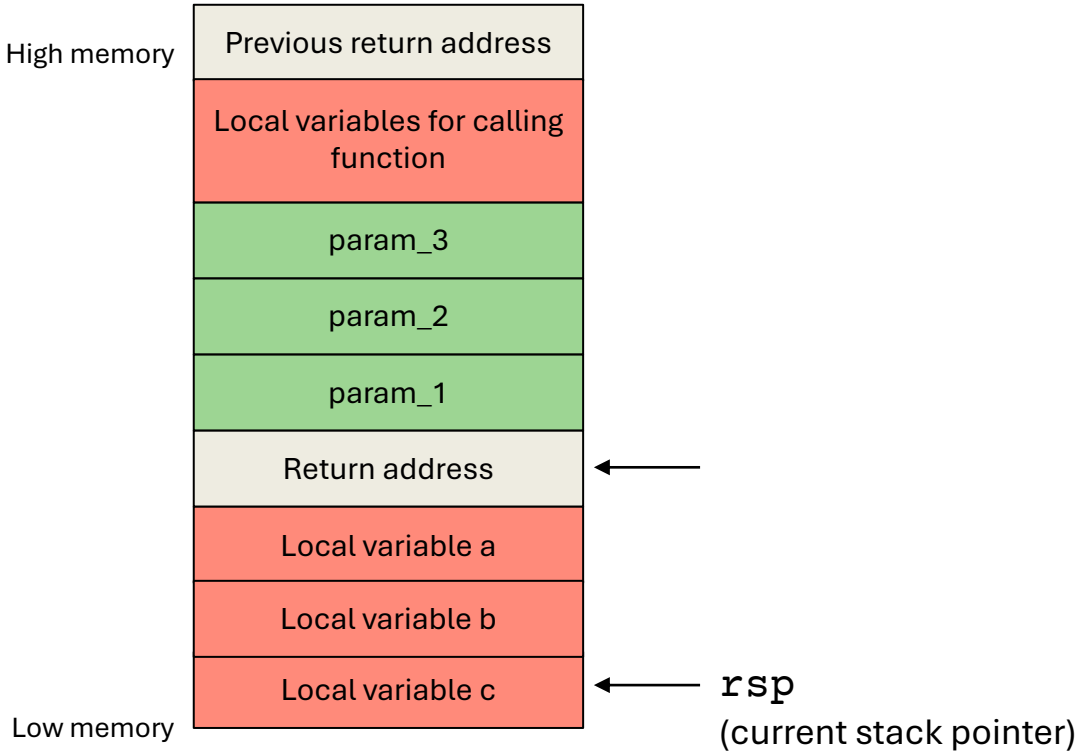
Loaded by *execve*

Note:
The **stack** stores local variables, saved registers, and return addresses. It grows **downward**.

The **heap** stores dynamically-allocated data. It grows **upward**.

*Not to scale

The stack




```
func(param_1, param_2, param_3)

Calling function:
        pushl param_3
        pushl param_2
        pushl param_1
        call func
        . . .

Called function:
func:   subl $12, %rsp
        . . .
        addl $12, %rsp
        ret
```

Causing overflow

Overflow can occur when programs do not validate the length of data being written to a buffer

These "unsafe" functions could be in your code or hidden in libraries

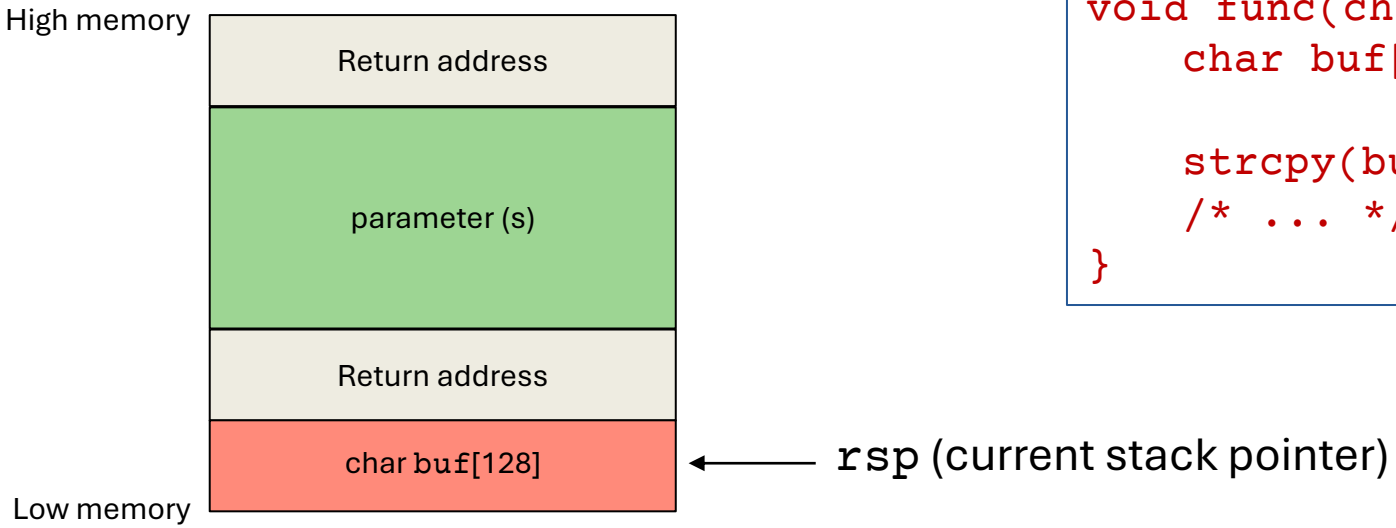
- `strcpy(char *dest, const char *src);`
- `strcat(char *dest, const char *src);`
- `gets(char *s);`
- `scanf(const char *format, …)`
- Others…

Safe functions

Counterparts that take a count as a parameter

```
strncpy(dest, src, count)
strncat(dest, src, count)
fgets(buf, file, count)
Sscanf(format, count, ...)
```

Overflowing the buffer




```
void func(char *s) {
    char buf[128];

    strcpy(buf, s);
    /* ... */
}
```

What if `strlen(s)` is >127 bytes?

You overwrite the *return address*

Overwriting the return address

- If we overwrite the return address
  - We change what the program executes when it returns from the function
- "Benign" overflow
  - Overflow with garbage data
  - Chances are that the return address will be invalid
  - Program will die with a SEGFAULT
  - Availability attack

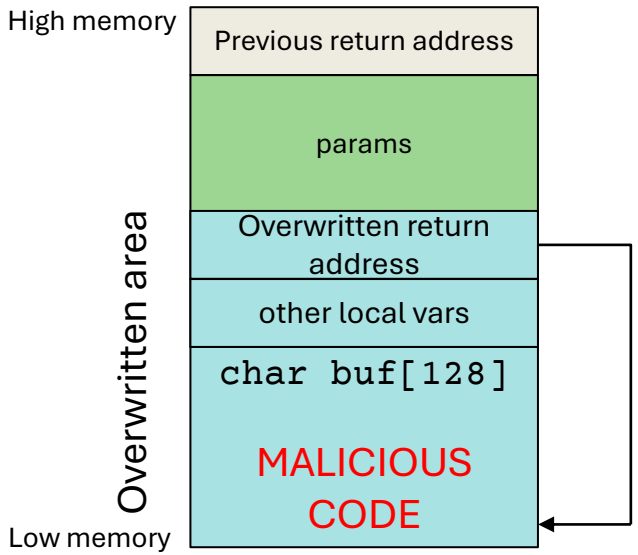
Programming at the machine level

- High level languages (even C) constrain you in
  - Access to variables (local vs. global)
  - Control flows in predictable ways
    - Loops, function entry/exit, exceptions
- At the machine code level
  - No restriction on where you can jump
    - Jump to the middle of a function … or to the middle of a C statement
    - Returns will go to whatever address is on the top of the stack
    - Unused code can be executed (e.g., library functions not used by the program)

Subverting control flow

Malicious overflow

- Fill the buffer with malicious code
- Then overwrite saved the stack pointer (the return address) with the address of the malicious code in the buffer

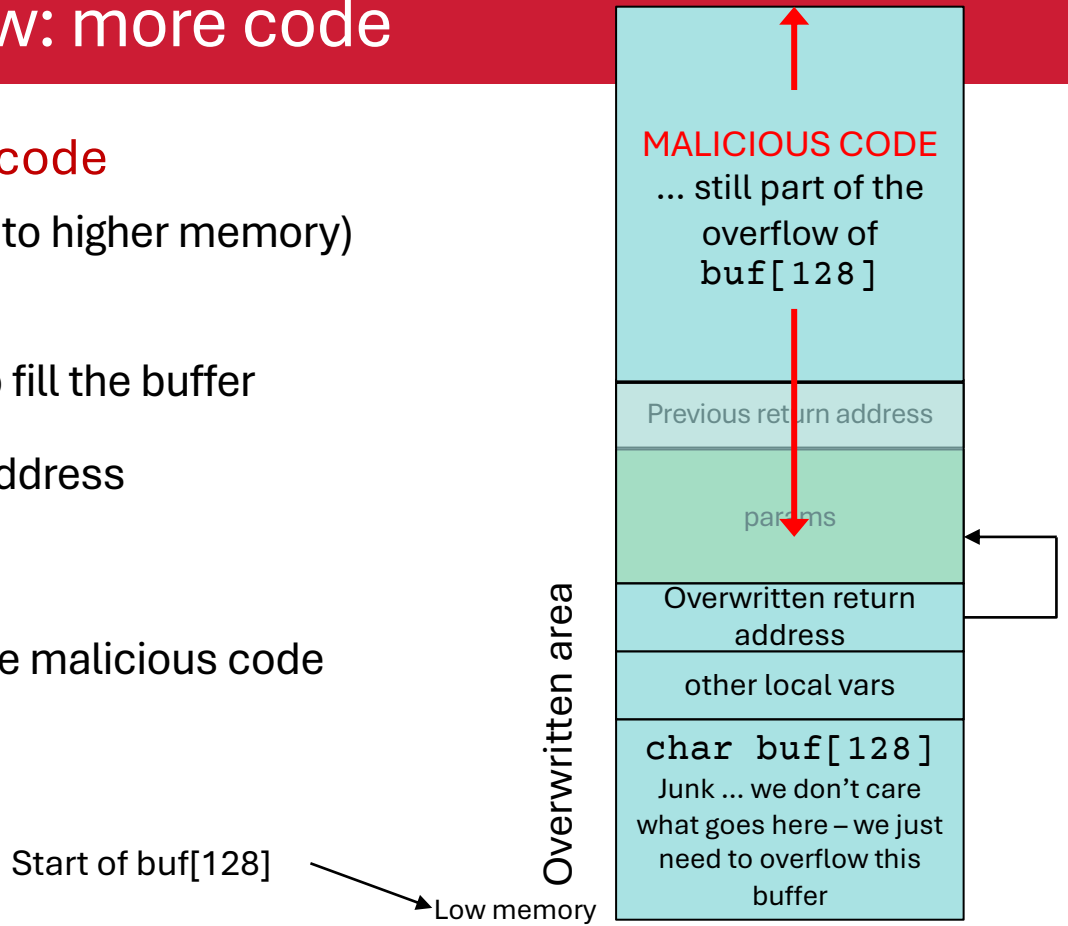


Subverting control flow: more code

If you want to inject a lot of code

Just go further down the stack (into higher memory)

- Initial parts of the buffer will be garbage data … we just need to fill the buffer
- Then we have the new return address
- Then we have malicious code
- The return address points to the malicious code

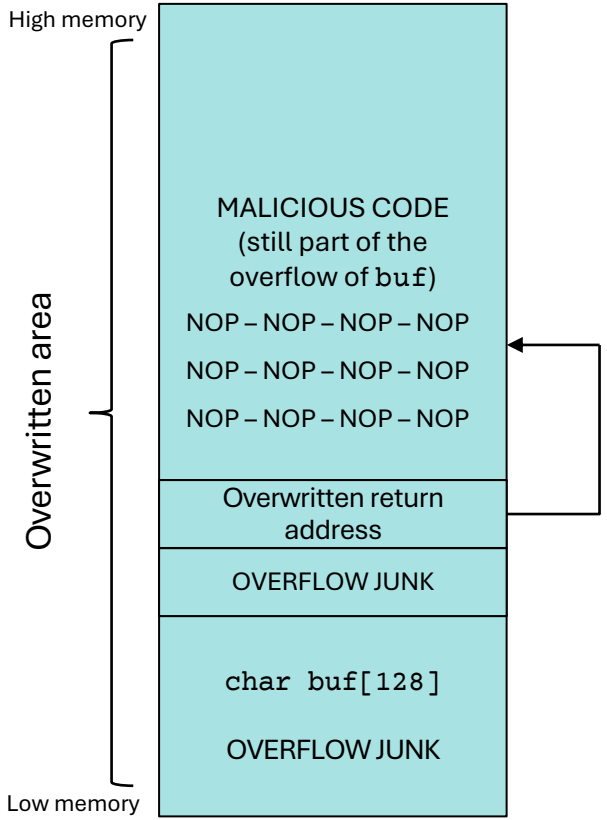


Address Uncertainty

What if we're not sure what the exact address of our injected code is?

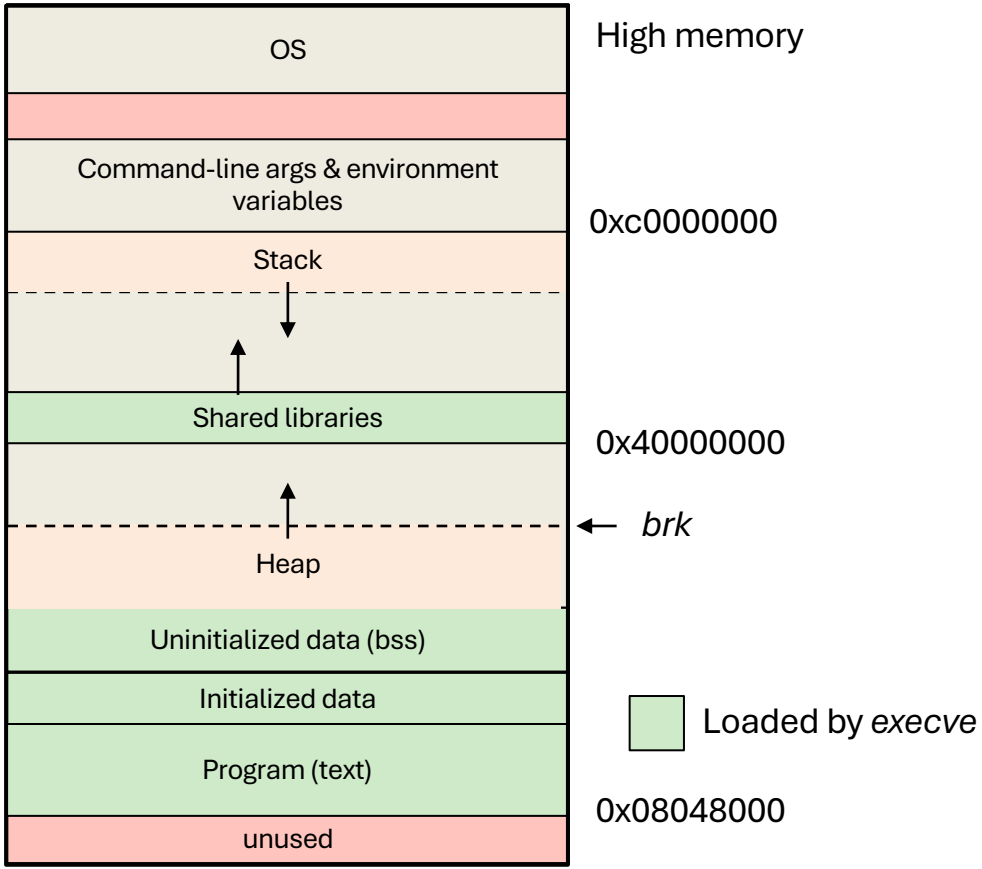
NOP slide = NOP sled = landing zone

- Pre-pad the code with lots of NOP instructions
  - NOP
  - moving a register to itself
  - Oring a register with itself
  - etc.
- Set the return address on the stack to any address within the landing zone



Heap & text overflows

Linux process memory map



- Statically allocated variables & dynamically allocated memory (*malloc*) are not on the stack
- Heap data & static data do not contain return addresses
  - No ability to overwrite a return address

Are we safe?

Memory overflow

We may be able to overflow a buffer and overwrite other variables in *higher* memory.

For example, overwrite the value of another string.

The program

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

char a[15];
char b[15];

int
main(int argc, char **argv)
{
  strcpy(b, "abcdefghijklmnopqrstuvwxyz");
  printf("a=%s\n", a);
  printf("b=%s\n", b);
  exit(0);
}
```

The output
(Linux 4.4.0-59, gcc 5.4.0)

```
a=qrstuvwxyz
b=abcdefghijklmnopqrstuvwxyz
```

Memory overflow – filename example

The program

We overwrote the file name `afile` by writing too much into `mybuf`!

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

char afile[20];
char mybuf[15];

int main(int argc, char **argv)
{
     strncpy(afile, "/etc/secret.txt", 20);
     printf("Planning to write to %s\n", afile);
     strcpy(mybuf, "abcdefghijklmnop/home/paul/writehere.txt");
     printf("About to open afile=%s\n", afile);
     exit(0);
}
```

mybuf can overflow into `afile`

The output
(Linux 5.10.63, gcc 8.3.0)

```
Planning to write to /etc/secret.txt
About to open afile=/home/paul/writehere.txt
```

Overwriting variables: changing control flow

Even if a buffer overflow does not touch the stack, it can modify global or static variables – or dynamically-allocated content

- Example:
  - Overwrite a function pointer
    - Function pointers are often used in callbacks
    - C++ builds tables of function pointers to implement polymorphism
  - Overwrite variables that affect control flow

```
int callback(const char* msg)
{
     printf("callback called: %s\n", msg);
}

int main(int argc, char **argv)
{
     static int (*fp)(const char *msg);
     static char buffer[16];

     fp = (int(*)(const char *msg))callback;
     strcpy(buffer, argv[1]);
     (int)(*fp)(argv[2]);     // call the callback
}
```

The exploit

- The program takes the first two arguments from the command line
- It copies `argv[1]` into a buffer with no bounds checking
- It then calls the callback, passing it the message from the 2nd argument

The exploit

- Overflow the buffer
- The overflow bytes will contain the address you really want to call
  - They're strings, so bytes with 0 in them will not work … which is a common challenge with many string-based attacks

```
int callback(const char* msg)
{
     printf("callback called: %s\n", msg);
}

int main(int argc, char **argv)
{
     static int (*fp)(const char *msg);
     static char buffer[16];

     fp = (int(*)(const char *msg))callback;
     strcpy(buffer, argv[1]);
     (int)(*fp)(argv[2]);     // call the callback
}
```

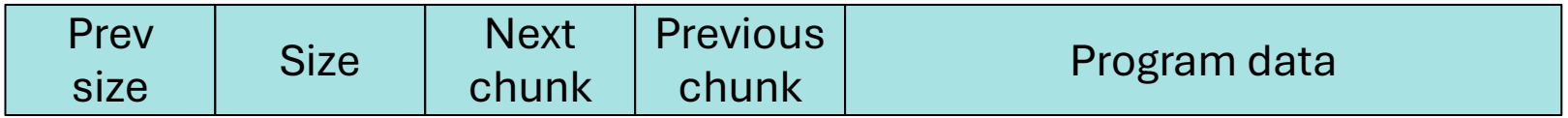
Use-after-free, Double-free attacks

- **Use-after-free**: keep using a reference after freeing it
  - Memory that is freed will get allocated new memory allocations
  - If the original reference is still used, an attacker may change the contents
  - Hijack vtables (tables of pointers to virtual functions to support polymorphism) or pointer tables
  - Read sensitive data
- **Double-free**: can corrupt a memory allocator's data structures

Memory can be reallocated to something that the attacker can control (user input or buffer overflow or some other vulnerability)

Heap corruption

- Dynamically-allocated memory (via *new* or *malloc*) is managed by a **memory allocator** – a library that asks the kernel for memory and then reuses free space
- Overflows can corrupt the metadata that keeps track of how memory is allocated
  - The details depend on the design of the memory allocator



- Basic defense: sanity check
  - Check that the pointer to the next chunk is in the heap
  - Check that the *previous chunk* of the next chunk points to this chunk
- But the check can only take place when there's a call to the allocator

Part 3

Integer Overflow

Minimum & maximum values for integers

Size	Unsigned	Signed
8-bit (1 byte)	0 .. 255	-128 .. +127
16-bit (2 bytes)	0 .. 65,535	-32,768 .. +32767
32-bit (4 bytes)	0 .. 4,294,967,295	-2,147,483,648 .. 2,147,483,647
64-bit (8 bytes)	0 .. 18,446,744,073,709,551,615	-9,223,372,036,854,775,808 .. +9,223,372,036,854,775,807

- Arbitrary precision libraries may be available
  - But processors don't do arbitrary precision math, so there's a performance penalty

Overflows and underflows

Going outside the range causes an overflow or underflow

- No room for the extra bit
- These do not generate exceptions

```
  11111111
+ 00000001
 100000000
```

255 + 1 = 0

Unsigned integer overflow

Bigger than the biggest?

```
int main(int argc, char **argv)
{
    unsigned short n = 65535;

    printf("n = %d\n", n);
    n = n + 1;
    printf("n+1 = %d\n", n);
}
```

max unsigned short int

What gets printed?

```
n = 65535
n+1 = 0
```

Signed integer overflow

Bigger than the biggest?

```
int main(int argc, char **argv)
{
    short n = 32767;

    printf("n = %d\n", n);
    n = n + 1;
    printf("n+1 = %d\n", n);
}
```

max short int

What gets printed?

```
n = 32767
n+1 = -32768
```

Also underflow

Smaller than the smallest?

```
int main(int argc, char **argv)
{
    short n = -32768;

    printf("n = %d\n", n);
    n = n - 1;
    printf("n-1 = %d\n", n);
}
```

max short int

What gets printed?

```
n = -32768
n-1 = 32767
```

Same thing for ints

Bigger than the biggest?

```
int main(int argc, char **argv)
{
    short n = 2147483647;

    printf("n = %d\n", n);
    n = n + 1;
    printf("n+1 = %d\n", n);
}
```

max int

What gets printed?

```
n = 2147483647
n+1 = -2147483648
```

Signed/unsigned mismatches

Casting from unsigned to signed

```
int main(int argc, char **argv)
{
    unsigned short n = 65535;
    short i = n;

    printf("n = %d\n", n);
    printf("i = %d\n", i);
}
```

What gets printed?

```
n = 65535
i = -1
```

So what?

You might not detect a buffer overflow because of an integer overflow

- Processors don't generate exceptions for overflows/underflows

If *packet_get_int* returns 1073741824
and **`sizeof(char*)`** = 4,
we allocate 0 bytes for response!

Version 3.3 of OpenSSH

```
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i < nresp; i++)
  response[i] = packet_get_string(NULL);
}
```

But we have 64-bit architectures!

- Even 64-bit values can overflow

  If users can set a field to any value, they can set it to a huge value and overflows can occur

- Default `int` size in C on Windows, Linux, macOS = 32 bits

- A lot of data fields in network messages use smaller values

IP header	Time-to-live Fragment offset Length	8 bits 16 bits 16 bits
TCP header	Sequence #, ack # Window size	32 bits 16 bits
GPS info	Week #	10 bits

Some values are constrained

A lot of data fields in network messages use smaller values

- IP header
  - time-to-live field = 8 bits, fragment offset = 16 bits, length = 16 bits
- TCP header
  - Sequence #, Ack # = 32 bits, Window size = 16 bits
- GPS week # = 10 bits

Python 3 has no size limit

- Actual type is hidden from the user
  - Internally, an integer (32 or 64 bit, depending on the CPU) is used and is converted to an arbitrary-length integer object when needed
- But there's a cost!
  - 10B iterations of incrementing an int on an M2 Mac
    - C: 4.44 seconds
    - Java: 28.8 seconds – 6.4x slower
    - Python 237 seconds – 53x slower

By the way, do you trust Python's math?

```
$ python3 -q
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 - 0.3
5.551115123125783e-17
```

SOPHOS

Patch now! Microsoft releases fixes for the serious SMB bug CVE-2020-0796

March 12, 2020

...
The SMBv3 vulnerability fixed this month is a doozy: A potentially network-based attack that can bring down Windows servers and clients, or could allow an attacker to run code remotely simply by connecting to a Windows machine over the SMB network port of 445/tcp. The connection can happen in a variety of ways we describe below, some of which can be exploited without any user interaction.
...
Microsoft fixes 116 vulnerabilities with this month's patches, and considers 25 of them critical, and 89 important. All the critical vulnerabilities could be used by an attacker to execute remote code and perform local privilege elevation.

https://news.sophos.com/en-us/2020/03/12/patch-tuesday-for-march-2020-fixes-the-serious-smb-bug-cve-2020-0796/

2020 SMB Bug: CVE-2020-0796 (SMBGhost)

"The vulnerability involves an integer overflow and underflow in one of the kernel drivers. The attacker could craft a malicious packet to trigger the underflow and have an arbitrary read inside the kernel, or trigger the overflow and overwrite a pointer inside the kernel. The pointer is then used as destination to write data. Therefore, it is possible to get a write-what-where primitive in the kernel address space."

Bug in the compression mechanism of SMB in Windows 10

Attacker can control two fields

- `OriginalCompressedSegmentSize` and `Offset`
- Use a huge value for `OriginalCompressedSegmentSize` to cause overflow
  - This will cause the system to allocate fewer bytes than necessary
  - Decompress will cause an overflow

https://blog.zecops.com/research/exploiting-smbghost-cve-2020-0796-for-a-local-privilege-escalation-writeup-and-poc/

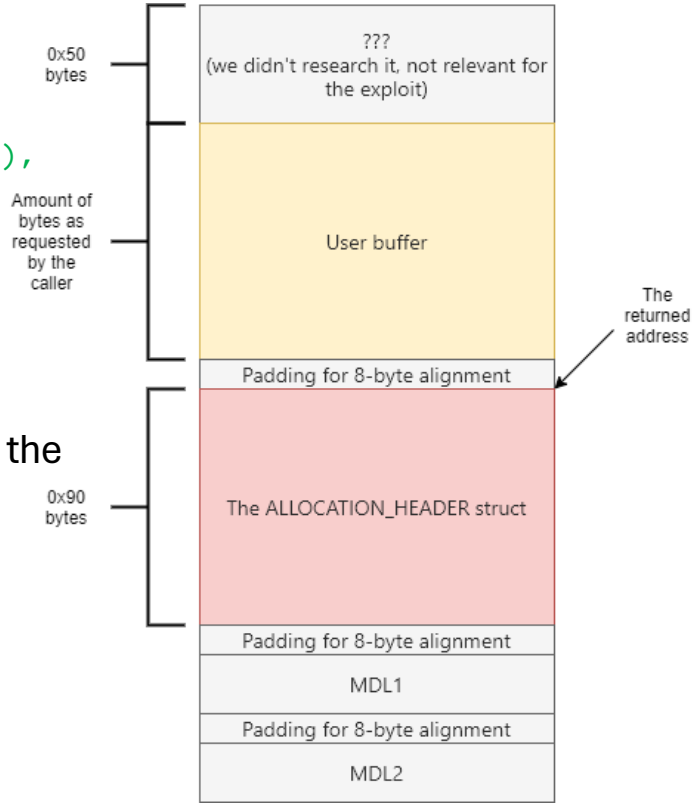
2020 SMB Bug: CVE-2020-0796 (SMBGhost)

Program does

```
memcpy( Alloc->UserBuffer,
     (PUCHAR)Header + sizeof(COMPRESSION_TRANSFORM_HEADER),
     Header->Offset);
```

Attack

- A decompression into a smaller buffer can overflow the User buffer
- The target of *memcpy* (`Alloc->UserBuffer`) is read from the allocation header, which can be overwritten
- The Header contents & offset can also be set by the attacker
- The attacker can write anything anywhere in kernel memory!



https://blog.zecops.com/research/exploiting-smbghost-cve-2020-0796-for-a-local-privilege-escalation-writeup-and-poc/

Microsoft Exchange year 2022 bug in FIP-FS breaks email delivery

BLEEPINGCOMPUTER

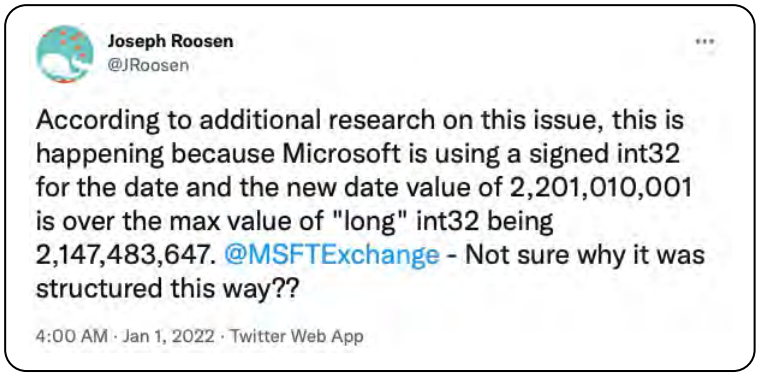
Lawrence Abrams • January 1, 2022

Microsoft Exchange on-premise servers cannot deliver email starting on January 1st, 2022, due to a "Year 2022" bug in the FIP-FS anti-malware scanning engine.

Starting with Exchange Server 2013, Microsoft enabled the FIP-FS anti-spam and anti-malware scanning engine by default to protect users from malicious email.

Microsoft Exchange Y2K22 bug

According to numerous reports from Microsoft Exchange admins worldwide, a bug in the FIP-FS engine is blocking email delivery with on-premise servers starting at midnight on January 1st, 2022.



https://www.bleepingcomputer.com/news/microsoft/microsoft-exchange-year-2022-bug-in-fip-fs-breaks-email-delivery/

Is .gif a GIF file? Assumptions about file formats

- iOS Messages app
  - Any embedded file with a `.gif` extension will be decoded before the message is shown
    - Sent to the *IMTranscoderAgent* process that uses the ImageIO library
    - The ImageIO library ignores the file name and tries to guess the format to parse it
  - Allows attackers to send files in over 20 formats, increasing the attack surface
- This was used in NSO's Pegasus malware on the iPhone
  - Zero-click install via iMessages
  - Sent a PDF file with a `.gif` file name
  - Contents were compressed with JBIG2 compression

See https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

PDF – JBIG2 Compression – Integer Overflow

- JBIG2 compression
  - Extreme compression format for black & white images
  - Breaks images into segments
  - Contains table with pointers to similar bitmaps
- This attack exploited an integer overflow bug
  - With carefully crafted segments, the count of detected symbols could overflow
  - This results in the allocated buffer being too small
  - Bitmaps are then written into this buffer
    (which is much too small, so the data ends up being written beyond the buffer)
  - Enables attacker to control what gets written into arbitrary memory

PDF – JBIG2 Compression – Integer Overflow

```
Guint numSyms; // (1)
numSyms = 0;
for (i = 0; i < nRefSegs; ++i) {
   if ((seg = findSegment(refSegs[i]))) {
      if (seg->getType() == jbig2SegSymbolDict) {
        numSyms += ((JBIG2SymbolDict *)seg)->getSize();  // (2)
      } else if (seg->getType() == jbig2SegCodeTable) {
        codeTables->append(seg);
      }
   } else {
      ...
...
// get the symbol bitmaps
  syms = (JBIG2Bitmap **)gmallocn(numSyms, sizeof(JBIG2Bitmap *)); // (3)
  kk = 0;
  for (i = 0; i < nRefSegs; ++i) {
    if ((seg = findSegment(refSegs[i]))) {
      if (seg->getType() == jbig2SegSymbolDict) {
        symbolDict = (JBIG2SymbolDict *)seg;
        for (k = 0; k < symbolDict->getSize(); ++k) {
          syms[kk++] = symbolDict->getBitmap(k); // (4)
        }
...
```

32-bit symbol count

Symbol count can overflow with too many segments. `numSyms` becomes a small #

Allocated buffer becomes too small

Off-by-one overflows

Miscounts happen

```
char buf[512];
int i;

for (i=0; i<=512; i++)
   buf[i] = stuff[i];
```

Miscounts happen

```
char buf[512];
int i;

for (i=0; i<=512; i++)
   buf[i] = stuff[i];
```

Safe functions require a count – but can't protect against mistakes

- *strcpy* → *strncpy*
- *strcat* → *strncat*
- *sprintf* → *snprintf*

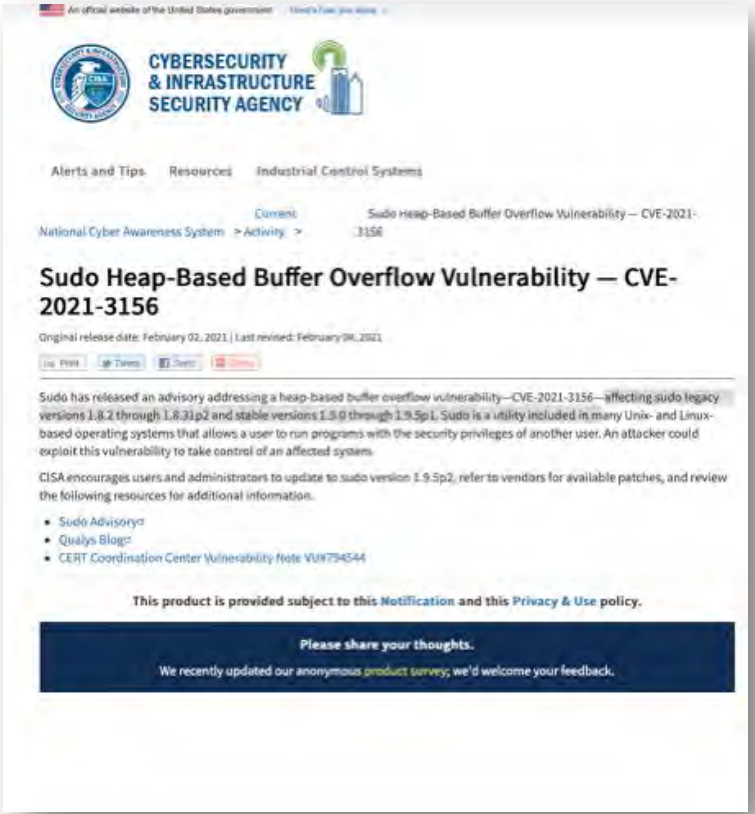
Off-by-one overflow

Feb. 2, 2021: Linux sudo

- Heap-based buffer overflow vulnerability
- An attacker could exploit this vulnerability to take control of an affected system.

- Off-by-one error
  - Can result in a heap-based buffer overflow, which allows privilege escalation to root via "`sudoedit -s`" and a command-line argument that ends with a single backslash character.

The parser would skip over the next character if it saw a `\`, skipping over the 0 byte that terminates a string



https://www.cisa.gov/uscert/ncas/current-activity/2021/02/02/sudo-heap-based-buffer-overflow-vulnerability-cve-2021-3156

Part 4

Format string/printf vulnerabilities

printf and its variants

Standard C library functions for formatted output

- printf: print to the standard output
- wprintf: wide character version of printf
- fprintf, wfprintf: print formatted data to a FILE stream
- sprintf, swprintf: print formatted data to a memory location
- vprintf, vwprintf, vfprintf, vwfprintf :
  print formatted data containing a pointer to argument list

Usage

```
printf(format_string, arguments ...)
printf("The number %d in decimal is %x in hexadecimal\n", n, n);
printf("my name is %s\n", name);
```

Dangerous usage of `printf`

Valid:

```
printf("hello, world!\n")
```

Valid, but more dangerous:

```
char *message = "hello, world\n";
printf(message);
```

Dangerous:

```
char *message = user_input();
printf(message);
```

It's safer to use `puts` or `fputs` if you don't need to parse formatting directives.

printf is dangerous if an attacker can change the format string. It allows:

- Leaking stack data
- Writing to memory

Reserve `printf` for places where formatting is really needed.

Leaking data: dumping memory with printf

```
$ ./tt hello
hello

$ ./tt "hey: %012lx"
hey: 7fffe14a287f
```

printf does not know how many arguments it has.
It deduces that from the format string.

If you don't give it enough, it keeps reading from the stack

We can dump arbitrary memory by walking up the stack

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

int show(char *buf)
{
    printf(buf); putchar('\n');
    return 0;
}

int main(int argc, char **argv)
{
    if (argc == 2)
        show(argv[1]);
}
```

```
$ ./tt 0x%08x.0x%08x.0x%08x.0x%08x.0x%08x
0x6ed0cf98.0x6ed0cfb0.0xd4ec1db8.0x17f4ff10.0x17f95040
```

Writing to memory with printf

Have you ever used `%n` ?

Format specifier that will store into memory the number of bytes written so far

```
int printbytes;
printf("paul%n says hi\n", &printbytes);
```

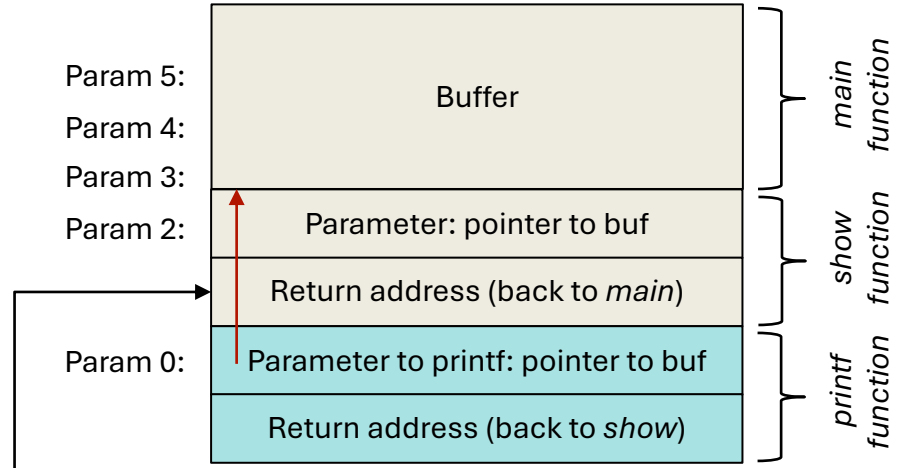
Will print

```
paul says hi
```

and will store the number `4` (which is the value of `strlen("paul")`) into the variable `printbytes`

If we combine this with the ability to change the format specifier, we can write to other memory locations

Writing to memory with %n




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

int
show(char *buf)
{
    printf(buf);
    putchar('\n');
    return 0;
}

int
main(int argc, char **argv)
{
    char buf[256];

    if (argc == 2) {
        strncpy(buf, argv[1], 255);
        show(buf);
    }
}
```

- *printf* treats this as the 1st parameter after the format string.
  - **Traverse the stack to get to the format string:**
    - We can skip parameters with formatting strings such as %x
    - That will make *printf* to read the next parameter from the format string
  - **Setting the value to be written with format specifiers**
    - Specify minimum widths to increase the byte count
  - **Writing the value**
    - The # of bytes output will be written to the attacker's chosen address, which will be the first 8 bytes of the format string

printf attacks: %n

What good is %n when it's just # of bytes written?

You can specify an arbitrary number of bytes in the format string

```
printf("\x02ABCDEFG%.08x%.622404x%.622400x%n" . . .
```

Traverse the stack to get to the format string:

- Each occurrance of %x (or %d, %b, …) will go down the stack by one parameter
- Each In this example, we go 3 parameters deep (e.g., 3*8 = 24 bytes)

Set the value to write by using field widths for outputs

- `%.622404x` = write 622404 characters for this value
- The sum of all the bytes output = the value that %n will write
- Value = 8+622404+622400 = 1244812 = 0x12fe8c

Write to the desired address

- printf expects the next item on the stack to be the address for %n: but it's our format string
- Set the first 8 bytes of the format string to be the address we want to change
  - We can use \NNN escapes to write non-printable characters
  - address = "\012ABCDEFG" = 0x0a41424344454647

-Wformat-security

Compiler flag to warn if `printf` and `scanf` use a format string that's not a string literal and there are no arguments to the format
But this doesn't help

Sanitize inputs

Don't use inputs blindly – check that they conform to a format you expect
(e.g., a name should not have a %n in it)

Printf attack mitigations

Part 5

Defending against hijacking attacks

Fix bugs

- Audit software
- Check for buffer lengths whenever adding to a buffer
- Search for unsafe functions
  - Use *nm* and *grep* to look for function names
- Use automated tools
  - Clockwork, CodeSonar, Coverity, Parasoft, PolySpace, Checkmarx, PREfix, PVS-Studio, PCPCheck, Visual Studio
- Most compilers and/or linkers now warn against bad usage

```
tt.c:7:2: warning: format not a string literal and no format arguments [-Wformat-security]
zz.c:(.text+0x65): warning: the 'gets' function is dangerous and should not be used.
```

Sanitizers

Compilers can add runtime checks for memory and math errors

- **AddressSanitizer (ASan)**
  Detects out-of-bounds and use-after-free errors
- **Undefined Behavior Sanitizer (UBSan)**
  Detects invalid operations such as signed integer overflow or bad type casts
- **LeakSanitizer**
  Reports unfreed memory and pointer leaks

Generate reproducible crashes when problems arise

But these add overhead, so are used in testing rather than production

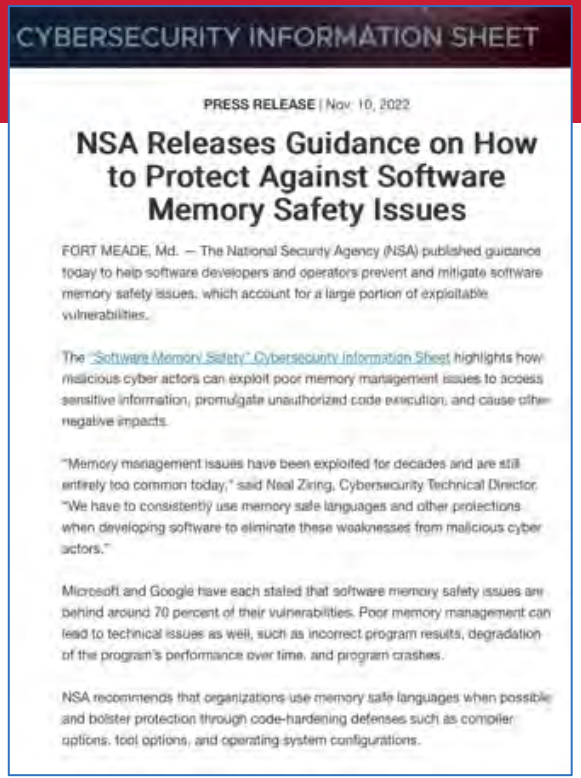
Fix bugs: Fuzzing

Do what the attackers do and try to locate unchecked assumptions!

- Generate semi-random data as input to detect bugs
  - Useful in locating input validation & buffer overflow problems
  - Enter unexpected input
  - See if the program crashes
- Enter long strings with searchable patterns
- If the app crashes
  - Search the core dump for the fuzz pattern to find where it died (or use a fuzzing tool)
- Automated fuzzer tools help with this
  - E.g., libFuzzer and AFL in C/C++; cargo-fuzz in Rust, Go Fuzzing
- Or … try to construct exploits using gdb

Don’t use C or C++

- Most other languages feature
  - Run-time bounds checking
  - Parameter count checking
  - Disallow reading from or writing to arbitrary memory locations
- Hard to avoid in many cases
  - Lots of legacy code
  - Performance concerns, CPU load
  - Programmer skill, availability of libraries, long-term support
  - Top contenders: Rust and Go
    - Rust: created by Mozilla – Memory safety with the efficiency of C/C++
    - Go: created by Google – fast, compiled code
    - Go designed for faster compilation, Rust is designed for faster execution



CYBERSECURITY INFORMATION SHEET

PRESS RELEASE | Nov 10, 2022

NSA Releases Guidance on How to Protect Against Software Memory Safety Issues

FORT MEADE, Md. — The National Security Agency (NSA) published guidance today to help software developers and operators prevent and mitigate software memory safety issues, which account for a large portion of exploitable vulnerabilities.

The "Software Memory Safety" Cybersecurity Information Sheet highlights how malicious cyber actors can exploit poor memory management issues to access sensitive information, promulgate unauthorized code execution, and cause other negative impacts.

"Memory management issues have been exploited for decades and are still entirely too common today," said Neal Ziring, Cybersecurity Technical Director. "We have to consistently use memory safe languages and other protections when developing software to eliminate these weaknesses from malicious cyber actors."

Microsoft and Google have each stated that software memory safety issues are behind around 70 percent of their vulnerabilities. Poor memory management can lead to technical issues as well, such as incorrect program results, degradation of the program's performance over time, and program crashes.

NSA recommends that organizations use memory safe languages when possible and bolster protection through code-hardening defenses such as compiler options, tool options, and operating system configurations.

https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/

Don’t use C or C++

- Google’s switch to memory-safe languages led to the % of memory-safe vulnerabilities in Android dropping from 76% to 24% over six years.
- Google announced support for Rust in Android in 2021

The Hacker News

Google's Shift to Rust Programming Cuts Android Memory Vulnerabilities by 68%

Sep 25, 2024 Ravie Lakshmanan

Google has revealed that its transition to memory-safe languages such as Rust as part of its secure-by-design approach has led to the percentage of memory-safe vulnerabilities discovered in Android dropping from 76% to 24% over a period of six years.

The tech giant said focusing on Safe Coding for new features not only reduces the overall security risk of a codebase, but also makes the switch more "scalable and cost-effective."

Eventually, this leads to a drop in memory safety vulnerabilities as new memory unsafe development slows down after a certain period of time, and new memory safe development takes over, Google's Jeff Vander Stoep and Alex Rebert said in a post shared with The Hacker News.

Perhaps even more interestingly, the number of memory safety vulnerabilities tends to register a drop notwithstanding an increase in the quantity of new memory unsafe code.

https://thehackernews.com/2024/09/googles-shift-to-rust-programming-cuts.html

Don’t use C or C++

- White House Office of the National Cyber Director called on developers to use languages without memory safety vulnerabilities

InfoWorld

by Grant Gross

White House urges developers to dump C and C++

News

Feb 27, 2024 · 5 mins

Biden administration calls for developers to embrace memory-safe programing languages and move away from those that cause buffer overflows and other memory access vulnerabilities.

CREDIT: MAGDALENA PETROVA

US President Joe Biden's administration wants software developers to use memory-safe programming languages and ditch vulnerable ones like C and C++.

The White House Office of the National Cyber Director (ONCD), in a report released Monday, called on developers to reduce the risk of cyberattacks by using programming languages that don't have memory safety vulnerabilities. Technology companies "can prevent entire classes of vulnerabilities from entering the digital ecosystem" by adopting memory-safe programming languages, the White House said in a news release.

https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

https://www.infoworld.com/article/2336216/white-house-urges-developers-to-dump-c-and-c.html

Don’t use C or C++

The Register®

DARPA suggests turning old C code automatically into Rust – using AI, of course

Who wants to make a TRACTOR pull request?

Thomas Claburn Sat 3 Aug 2024 10:03 UTC

To accelerate the transition to memory safe programming languages, the US Defense Advanced Research Projects Agency (DARPA) is driving the development of TRACTOR, a programmatic code conversion vehicle.

The term stands for TRanslating All C TO Rust. It's a DARPA project that aims to develop machine-learning tools that can automate the conversion of legacy C code into Rust.

https://www.theregister.com/2024/08/03/darpa_c_to_rust/

Ongoing attempts to fix C/C++

- **Safe C++ Extensions** proposal for inclusion in the C++ standard
  - Separate the safe and unsafe parts clearly – keep the safe parts useful
  - Don't break existing code
  - Addresses these categories of safety:
    - Lifetime safety (preserve objects with references), type safety (initialized vs. uninitialized data),
    - Thread safety (synchronization objects aren't opt-in), runtime checks (array bounds, bad division, bad references)
  - Safe Standard Library: Memory-safe implementations of essential algorithms

- **TrapC** – A proposed fork of C
  - Removes `goto` and `union`
  - Adopts a few C++ features that improve safety: Constructors & destructors, member functions
  - Automatic memory management
  - Limited lifetime for pointers

TrapC: https://www.infoworld.com/article/3836025/trapc-proposal-to-fix-c-c-memory-safety.html
Safe C++: https://safecpp.org/draft.html

Specify & test code

- If it's in the specs, it is more likely to be coded & tested
- Document acceptance criteria
  - "File names longer than 1024 bytes must be rejected"
  - "User names longer than 32 bytes must be rejected"
- Use safe functions that check & allow you to specify buffer limits
- Ensure consistent checks to the criteria across entire source
  - Example, you might `#define` limits in a header file but some files might use a mismatched number.
- Don't allow user-generated format strings and check results from *printf*

Safer libraries

- Compilers warn against unsafe *strcpy* or *printf*
- Ideally, fix your code!
- Sometimes you can't recompile (e.g., you lost the source)
- `libsafe`
  - Dynamically loaded library
  - Intercepts calls to unsafe functions
  - Validates that there is sufficient space in the current stack frame

```
(framepointer - destination) > strlen(src)
```

Dealing with buffer overflows: No Execute (NX)

Data Execution Prevention (DEP)

- Disallow code execution in data areas – on the stack or heap
- Set MMU per-page *execute* permissions to *no-execute*
- Intel and AMD added this support in 2004

Used in Windows, Linux, and macOS

No Execute – not a complete solution

No Execute Doesn't solve all problems

- Some legacy applications need an executable stack
- Some applications need an executable heap
  - code loading/patching
  - JIT (just-in-time) compilers
- NX does not protect against heap & function pointer overflows
- NX does not protect against `printf` and related format string problems

Return-to-libc

- Allows bypassing need for non-executable memory
  - With DEP, we can still corrupt the stack … just not execute code from it
- No need for injected code
- Instead, reuse functionality within the exploited app
- Use a buffer overflow attack to create a fake frame on the stack
  - Transfer program execution to a library function, running with the "restored" frame pointer
  - `libc` = standard C library … every program uses it!
  - Most common library function to exploit: `system`
    - Runs the shell with a specified command
    - New frame in the buffer contains a pointer to the command to run (which is also in the buffer)
      - E.g., `system("/bin/sh")`

Return Oriented Programming (ROP)

Generalize return-to-libc:

- Overwrite the return address to branch inside some library function
  - Does not have to be the start of the library routine
    - Use "borrowed chunks" of code from various libraries
  - When the library gets to a RET instruction, that location is on the stack, under the attacker's control
- Chain together sequences of code ending in RET
  - Build together "gadgets" for arbitrary computation
  - Buffer overflow contains a sequence of addresses that direct each successive RET instruction
- An attacker can use ROP to execute arbitrary algorithms without injecting new code into an application
  - Removing dangerous functions, such as *system*, is ineffective
  - To make attacking easier: use a compiler that combines gadgets!
    - Example: ROPC – a Turing complete compiler, `https://github.com/pakt/ropc`

Dealing with buffer overflows & ROP: ASLR

Addresses of everything in the code were well known

- Dynamically-loaded libraries were loaded in the same place each time, as was the stack & memory-mapped files
- Well-known locations make them branch targets in a buffer overflow attack

Address Space Layout Randomization (ASLR)

- Position stack and memory-mapped files to random locations
- Position libraries at random locations
  - Libraries must be compiled to produce position-independent code
- Implemented in all modern operating systems
  - OpenBSD, Windows ≥Vista, Windows Server ≥2008, Linux ≥2.6.15, macOS, Android ≥4.1, iOS ≥4.3
- But … not all libraries (modules) can use ASLR
  - And it makes debugging difficult

Address Space Layout Randomization

- Entropy
  - How random is the placement of memory regions?
  - If addresses are not random enough then attackers can guess
- Examples
  - Linux Exec Shield
    - 19 bits of stack entropy, 16-byte alignment – resulted in > 500K positions
  - Windows 7
    - Only 8 bits of randomness for DLLs
      - Aligned to 64K page in a 16MB region: resulted in 256 choices – far too easy to try them all!
  - Windows 8 onward
    - 24 bits for randomness on 64-bit processors: >16M possible placements

ASLR is effective only if the entropy is high enough and no information leak exists to disclose addresses

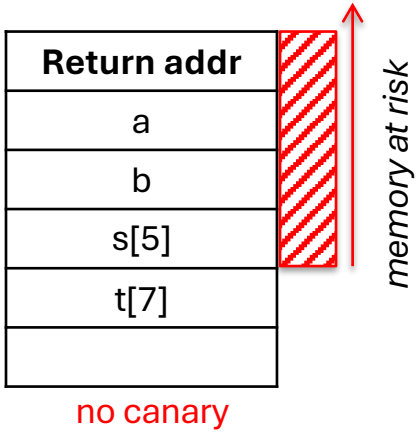
Dealing with buffer overflows: Canaries

Stack canaries

- Place a random integer before the return address on the stack
- Before a return, check that the integer is there and not overwritten: a buffer overflow attack cannot overwrite the return address without changing the canary

```
int a, b=999;
char s[5], t[7];

gets(s);
```



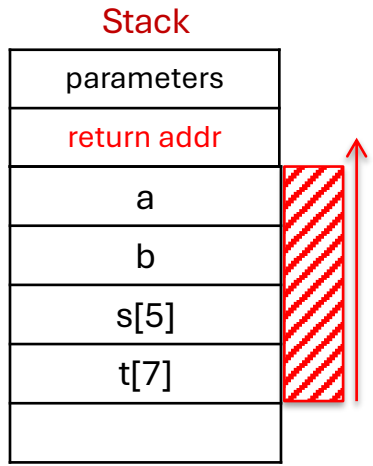
Dealing with buffer overflows: Canaries

Stack canaries

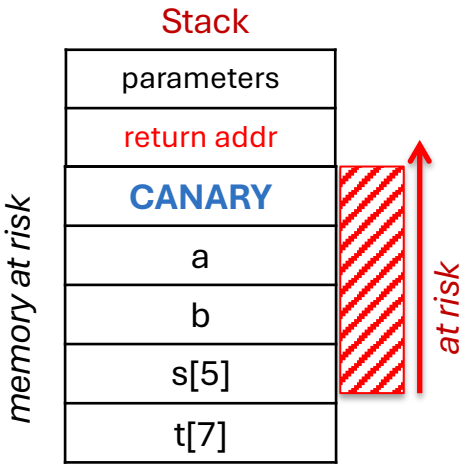
- Place a random integer before the return address on the stack
- Before a return, check that the integer is there and not overwritten: a buffer overflow attack cannot overwrite the return address without changing the canary

```
int a, b=999;
char s[5], t[7];

gets(s);
```



no canary



with canary

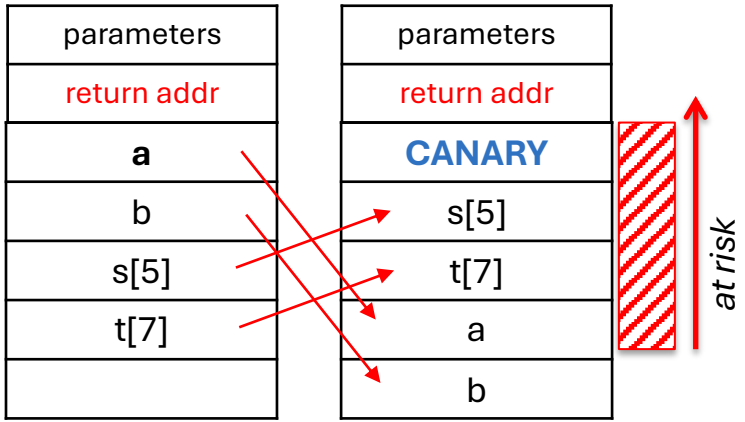
Refining Stack Canaries: Reordering Variables

IBM's ProPolice gcc patches – later incorporated into gcc

- Allocate local arrays into higher memory (below) other local variables in the stack
- Ensures that a buffer overflow attack will not clobber non-array variables
- Increases the likelihood that the overflow won't attack the logic of the current function

```
int a, b=999;
char s[5], t[7];

gets(s);
```



Stack canaries

- Not foolproof
- Heap-based attacks are still possible
- Performance impact
  - Need to generate a canary on entry to a function and check canary prior to a return
  - Minimal performance degradation ~8% for apache web server

Heap Protections (including use-after-free protection)

- Better code:
  - Don't use a reference after free
  - Set pointers to null after free
- Use a hardened allocator
  - isolates different types of memory allocation
  - Implements internal checks to detect memory corruption
  - Delays reallocation
- Heap canaries
  - Similar to stack canaries, an allocator inserts a secret value between allocated blocks and does a consistency check at various points to detect overwrites.

Heap Protections – Pointer Protection

- Encrypt pointers (especially function pointers)
  - Example: XOR with a stored random value
  - Any attempt to modify them will result in invalid addresses
  - XOR with the same stored value to restore original value
- Degrades performance when function pointers are used

Compiler-Based Control-Flow Integrity Checks

- Prevent attackers from hijacking a program's execution flow
- Control transfers (function calls, returns, indirect jumps)
  - Must follow legitimate paths defined in a control-flow graph at compile time
- Compiler inserts runtime checks before control transfers
  - If an attacker injects or overwrites a pointer to redirect execution to malicious code, the check will fail
- Primarily designed to block ROP

Intel CET: Control-Flow Enforcement Technology

Hardware support for Control-Flow Integrity

Developed by Intel & Microsoft to thwart ROP attacks

- Available starting with the Tiger Lake microarchitecture (mid-2020)

Two mechanisms

1. **Shadow stack**
   - Secondary stack
     - Only stores return addresses
     - MMU attribute disallows use of regular *store* instructions to modify the stack
   - Stack data overflows cannot touch the shadow stack – cannot change the control flow
2. **Indirect branch tracking**

Intel CET: Control-Flow Enforcement Technology

Indirect Branch Tracking

- Restrict a program's ability to use jump tables
- Jump table = table of memory locations the program can branch
  - Used for switch statements & various forms of lookup tables
- Prevent Jump-Oriented Programming (JOP) and Call Oriented Programming (COP)
  - Techniques where attackers abuse JMP or CALL instructions
  - Like Return-Oriented Programming but use gadgets that end with indirect branches
- New `ENDBRANCH` (ENDBR64) instruction allows a programmer to specify valid targets for indirect jumps
  - If you take an indirect jump, it has to go to an ENDBRANCH instruction
  - If the jump goes anywhere else, it will be treated as an invalid branch and generate a fault

ARM Heap Protection Mechanisms

Parallel mechanisms to Intel CET on ARM architectures:

ARM Pointer Authentication (PAC)

- Adds a pointer authentication code (like an HMAC) to verify pointers
- Requires compiler support
- Enabled by default in Linux

ARM Memory Tagging Extension (MTE)

- Each 16-byte block in memory gets a 4-bit tag.
- Pointers must have a matching tag in the unused upper bits of their address
- Guards against buffer overflows or use-after-free errors

ARM Heap Protection Mechanisms - Enhanced

Enhanced Memory Tagging Extension (EMTE)

- Developed jointly with Apple in 2022
- Accessing non-tagged memory (like global variables) from tagged regions requires attackers to know the region's tag

- Apple **Memory Integrity Enforcement** (MIE) – 2025
  - Memory allocators use type information to decide how to organize memory
  - Leverages EMTE
    - Memory tagging by allocator; adjacent allocations get different tags
    - Hardware enforces matching tag on access; tag mismatch triggers fault
    - Allocator retags memory regions when they get reused to detect use-after-free exploits
    - Kernel tag metadata protected; allocator retags on reuse to catch use-after-free

See: https://security.apple.com/blog/memory-integrity-enforcement/

Hardware Attacks: Example - Rowhammer

DDR4 memory protections are broken wide open by new Rowhammer technique

Researchers build "fuzzer" that supercharges potentially serious bitflipping exploits.

Dan Goodin • 11/15/2021

Rowhammer exploits that allow unprivileged attackers to change or corrupt data stored in vulnerable memory chips are now possible on virtually all DDR4 modules due to a new approach that neuters defenses chip manufacturers added to make their wares more resistant to such attacks.

Rowhammer attacks work by accessing—or hammering—physical rows inside vulnerable chips millions of times per second in ways that cause bits in neighboring rows to flip, meaning 1s turn to 0s and vice versa. Researchers have shown the attacks can be used to give untrusted applications nearly unfettered system privileges, bypass security sandboxes designed to keep malicious code from accessing sensitive operating system resources, and root or infect Android devices, among other things.

https://arstechnica.com/gadgets/2021/11/ddr4-memory-is-even-more-susceptible-to-rowhammer-attacks-than-anyone-thought/

Hardware Attacks: Example - Rowhammer

- RowHammer was disclosed in 2014
  - Exploits memory architecture to alter data by repeatedly accessing a specific row
  - This introduces random bit flips in neighboring memory rows
- 2021: new attack technique discovered
  - Uses non-uniform patterns that access two or more rows with different frequencies
  - Bypasses all defenses built into memory hardware
  - 80% of existing devices can be hacked this way
  - Cannot be patched!
- Sample attacks
  - Gain unrestricted access to all physical memory by changing bits in the page table entry
  - Give untrusted applications root privileges
  - Extract encryption key from memory

Fixed? Nope – introducing ZenHammer

- Manufacturers tried to mitigate this problem
- But in March, 2024…
  - Researchers created a new variant of the attack
  - ZenHammer acts like Rowhammer but can also flip bits on DDR5 devices

The Hacker News

New ZenHammer Attack Bypasses RowHammer Defenses on AMD CPUs

Mar 28, 2024 · Ravie Lakshmanan

Cybersecurity researchers from ETH Zurich have developed a new variant of the RowHammer DRAM (dynamic random-access memory) attack that, for the first time, successfully works against AMD Zen 2 and Zen 3 systems despite mitigations such as Target Row Refresh (TRR).

"This result proves that AMD systems are equally vulnerable to Rowhammer as Intel systems, which greatly increases the attack surface, considering today's AMD market share of around 36% on x86 desktop CPUs," the researchers said.

The technique has been codenamed ZenHammer, which can also trigger RowHammer bit flips on DDR5 devices for the first time.

RowHammer, first publicly disclosed in 2014, is a well-known attack that exploits DRAM's memory cell architecture to alter data by repeatedly accessing a specific row (aka hammering) to cause the electrical charge of a cell to leak to adjacent cells.

Summary of Mitigations

- Hardware
  - NX (no execute), CET (Intel Control-Flow Enforcement Technology), PAC (ARM Pointer Authentication Codes), MTE (ARM Memory Tagging Extensions)/EMTE (Enhanced MTE)
- OS/runtime
  - ASLR, hardened memory allocators, memory tagging
- Compiler/runtime
  - Stack canaries, safe compiler flags, control-flow integrity checks
- Developer
  - Bounds checking, safe APIs, sanitizers, fuzzing

Summary of Memory Attack Mitigations

Developer	Bounds checking Safe APIs, Sanitizers, fuzzing
Compiler/runtime	Stack canaries Safe compiler flags Control-flow integrity checks
OS/runtime	ASLR Hardened memory allocators Memory tagging (with hardware support)
Hardware	NX (no execute) CET (Intel Control-Flow Enforcement Technology), PAC (ARM Pointer Authentication Codes), MTE (ARM Memory Tagging Extensions)/EMTE (Enhanced MTE)

The end