Basic exploitation techniques

20210112

Outline

A primer on x86 assembly

Memory segments

Stack-based buffer overflows

Heap-based buffer overflows

Format strings

A primer on x86 assembly

Introduction

Verily, when the developer herds understand the tools that drive them to their cubicled pastures every day, then shall the 0day be depleted — but not before.
– Pastor Manul Laphroaig

It’s a trap!

- $\approx$ 1000 instructions . . .
- No time to know them all :-)

This overview is meant as a first help

Multiple syntaxes

- AT&T
- Intel

Basics

In general

Mnemonics accept from 0 to 3 arguments.

2 arguments mnemonics are of the form (Intel syntax)

$$m \text{ dst, src}$$

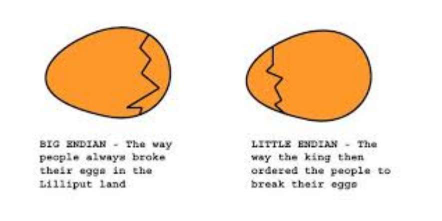
which roughly means

$$\text{dst} \leftarrow \text{dst} \odot \text{src}$$

where $\odot$ is the semantics of m

Endianness

`x = 0xdeadbeef`



Endianness

byte address	0x00	0x01	0x02	0x03
byte content (big-endian)	0xde	0xad	0xbe	0xef
byte content (litte-endian)	0xef	0xbe	0xad	0xde

Architectures

Big endian

PowerPC, Sparc, 68000

Little endian

Intel, AMD

Bi-endian

ARM, RISC-V

These usually defaults to little endian.

Resources

- Cheat sheet
- Opcode and Instruction Reference
- Intel full instruction set reference

Basic registers (16/32/64 bits)

64	32	16	name (8080) / use
r	e	ax	accumulator
r	e	bx	base address
r	e	cx	count
r	e	dx	data
r	e	di	source index
r	e	si	destination index
r	e	bp	base pointer
r	e	sp	stack pointer
r	e	ip	instruction pointer

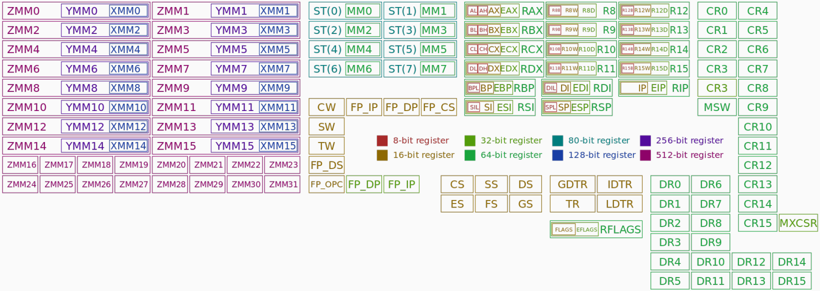
- `esp` (e = extended) is the 32 bits stack pointer
- `rsp` (r = register) is the 64 bits one

Less basic registers (64 bits)

Add extended general purpose registers r8-15

- r7*d* accesses the lower 32 bits of r7;
- r7*w* accesses the lower 16 bits;
- r7*b* accesses its lower 8 bits.

The full story



Register flags (partial)

of	overflow flag
cf	carry flag
zf	zero flag
sf	sign flag
df	direction flag
pf	parity flag
af	adjust flag

Signed vs unsigned

At machine-level, every value is a bitvector.

Bitvectors can be seen through different lenses:

- unsigned value
- signed value
- float (will not talk about it)

Transfer

Move

`mov dst, src`	dst := src
`xchg o1, o2`	tmp:= o1; o1 := o2; o2 := tmp

Arithmetic

All 4 arithmetic operations are present

`add src, dst`	dst ← dst + src
`sub src, dst`	dst ← dst - src
`div src`	t64 ← edx @ eax
	eax ← t64 / src
	edx ← t64 % src
`mul src`	t64 ← eax * src
	edx ← t64{32,63}
	eax ← t64{0,31}

Arithmetic

`inc dst`	dst ← dst + 1
`dec dst`	dst ← dst - 1
`sal/sar dst, src`	arithmetic shift left / right

Sign preservation

```
mov ax, 0xff00  # unsigned: 65280, signed : -256
# ax=1111.1111.0000.0000
sal ax, 2       # unsigned: 64512, signed : -1024
# ax=1111.1100.0000.0000
sar ax, 5       # unsigned: 65504, signed : -32
# ax=1111.1111.1110.0000
```

Basic logical operators

Basic semantics

`and`	`dst, src`	dst ← dst & src
`or`	`dst, src`	dst ← dst | src
`xor`	`dst, src`	dst ← dst ^ src
`not`	`dst`	dst ← ~dst

Examples

```
xor ax, ax        # ax = 0x0000
not ax            # ax = 0xffff
mov bx, 0x5500    # bx = 0x5500
xor ax, bx        # ax = 0xbbff
```

Logical shifts

Shift

`shl dst, src`	logical shift left
`shr dst, src`	logical shift right

Logical and arithmetic shift lefts are the same.

Example

```
mov ax, 0xff00  # unsigned: 65280, signed : -256
# ax=1111.1111.0000.0000
shl ax, 2       # unsigned: 64512, signed : -1024
# ax=1111.1100.0000.0000
shr ax, 5       # unsigned: 2016, signed : 2016
# ax=0000.0111.1110.0000
```

Comparison and test instructions

Comparison

`cmp dst, src` : set condition according to *dst* − *src*

Test

`test dst, src`: set condition according to *dst* & *src*

Stack manipulation

Push

```
push src    dec sp; @[sp] := src
```

Pop

```
pop src    src := @[sp]; inc sp
```

Nops

The `nop` instruction does nothing (it's skip!).

There are lots of `nop` instructions.

Assembly	Byte sequence
66 NOP	66 90H
NOP DWORD ptr [EAX]	0F 1F 00H
NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Misc

Lea (load effective address)

`lea dst, [src]`	dst := src
`mov dst, [src]`	dst := @[src]

Int

`int n` runs interrupt number *n*

Unconditional jump instructions

Call

```
call address
call *op
```

`call` pushes `eip`

Jmp

```
jmp *op
jmp address
```

`jmp` only jumps

Extra jumps

Leave

```
esp := ebp; ebp := pop();
```

Ret

```
esp := esp + 4; eip := @[esp - 4];
```

Unsigned jumps

jump	if	`n` version	`e` version
`ja`	above	✔	✔
`jb`	below	✔	✔
`jc`	carry	✔	✘

Reading

`ja` has *n* and *e* versions, means that mnemonics

- `jna` (not above),
- `jae` (above or equal),
- `jnae` (not above or equal)

exist as well

Signed jumps

jump type	if	`n` version	`e` version
`jg`	greater	✓	✓
`jl`	lower	✓	✓
`jo`	overflow	✓	✗
`js`	if sign	✓	✗

Addressing modes

The addressing mode determines, for an instruction that accesses a memory location, how the address for the memory location is specified.

Mode	Intel
Immediate	mov ax, 16h
Direct	mov ax, [1000h]
Register Direct	mov bx, ax
Register Indirect (indexed)	mov ax, [di]
Based Indexed Addressing	mov ax, [bx + di]
Based Indexex Disp.	mov eax, [ebx + edi + 2]

The semantics of instructions

may seem intuitive

but is complex

Instructions do have side effects

```
// 04 16 / add al, 0x16
 0: res8 := (eax(32){0,7} + 22(8))
 1: OF := ((eax(32){0,7}{7} = 22(8){7}) &
             (eax(32){0,7}{7} != res8(8){7}))
 2: SF := (res8(8) <s 0(8))
 3: ZF := (res8(8) = 0(8))
 4: AF := ((extu eax(32){0,7}{0,7} 9) + 22(9)){8}
 5: PF := !
           (((((((res8(8){0} ^ res8(8){1}) ^ res8(8){2}) ^
                  res8(8){3}) ^ res8(8){4}) ^ res8(8){5}) ^
               res8(8){6}) ^ res8(8){7}))
 6: CF := ((extu eax(32){0,7} 9) + 22(9)){8}
 7: eax{0, 7} := res8(8)
```

Real behavior of conditions

Mnemonic	Flag	`cmp x y`	`sub x y`	`test x y`
`ja`	$\neg$ CF $\wedge \neg$ ZF	$x >_u y$	$x' \neq 0$	$x \,\&y \ \neq 0$
`jnae`	CF	$x <_u y$	$x' \neq 0$	$\bot$
`je`	ZF	$x = y$	$x' = 0$	$x \,\&\, y = 0$
`jge`	OF = SF	$x \geq y$	$\top$	$x \geq 0 \vee y \geq 0$
`jle`	ZF $\vee$ OF $\neq$ SF	$x \leq y$	$\top$	$x\&y = 0 \vee$ $(x < 0 \wedge y < 0)$

Shift left

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is undefined.

Memory segments

General overview

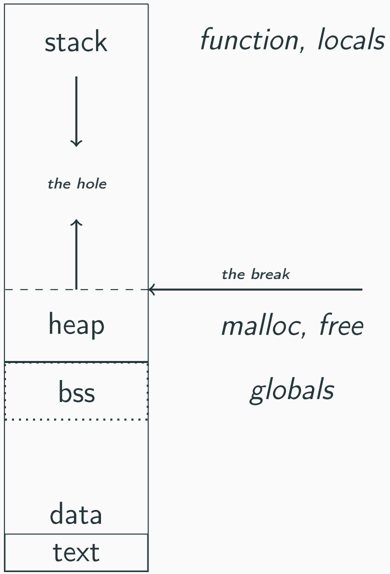
A compiled program has 5 segments:

1. code (text)
2. stack
3. data segments
   3.1 data
   3.2 bss
   3.3 heap

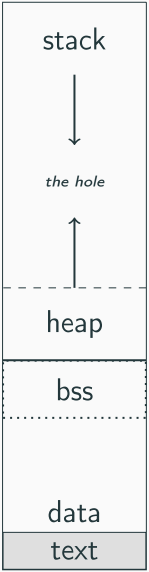
Execution

1. Read instruction *i* @ eip
2. Add byte length of *i* to eip
3. Execute *i*
4. Goto 1

Graphically speaking

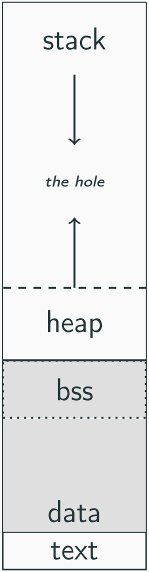


Text segment



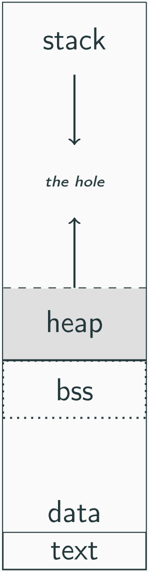
- The text segment (aka code segment) is where the code resides.
- It is not writable. Any attempt to to write to it will kill the program.
- As it is *ro*, it can be shared among processes.
- It has a fixed size

Data & bss segments



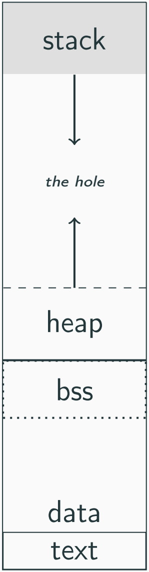
- The data segment is filled with initialized global and static variables.
- The bss segment contains the uninitialized ones. It is zeroed on program startup.
- The segments are (of course) writable.
- They have a fixed size

Heap segment



- The heap segment is directly controlled by the programmer
- Blocks can be allocated or freed and used for anything.
- It is writable
- It can grow larger, towards higher memory addresses – or smaller, on need

Stack segment



- The stack segment is a temporary scratch pad for functions
- Since eip changes on function calls, the stack is used to remember the previous state (return address, calling function base, arguments, . . .).
- It is writable
- It can grow larger, towards lower memory addresses – w.r.t to function calls.

In C

```c
void test_function(int a, int b, int c, int d)
{
   int flag;
   char buffer[10];
   flag = 31337;
   buffer[0] = 'A';
}

int main()
{
     test_function(1, 2, 3, 4);
}
```

Stack-based buffer overflows

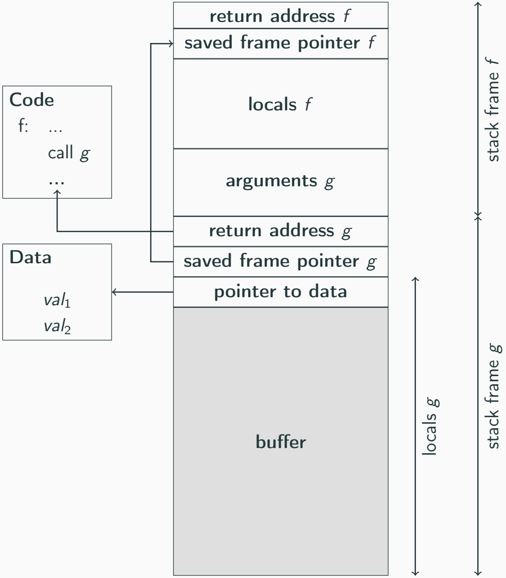
C low-level responsibility

In C, the programmer is responsible for data integrity.

This means there are no guards to ensure data is freed, or that the contents of a variable fits into memory,

This exposes memory leaks and buffer overflows

Reminder : stack layout for x86



Vulnerability reason

- When an array a is declared in C, space is reserved for it.
- a will be manipulated through offsets from its base pointer.
- At run-time, no information about the array size is present
- Thus, it is allowed to copy data beyond the end of a

A rich history

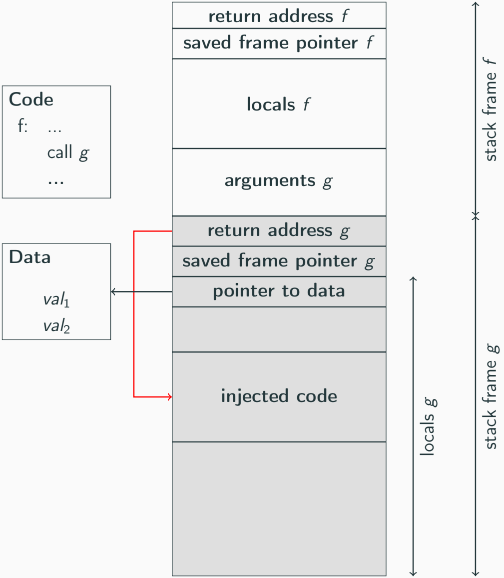
1972 First document attack

1988 Morris worm

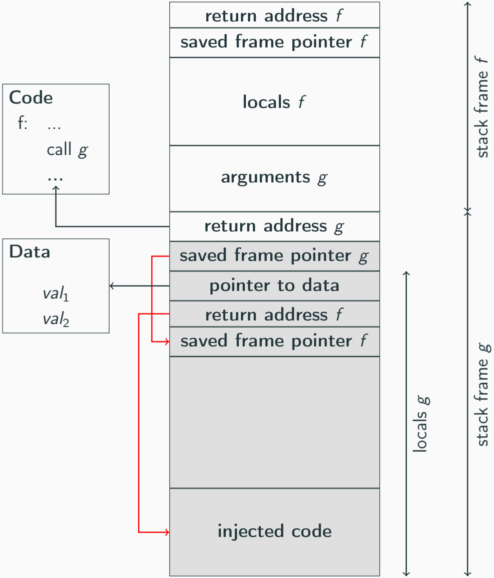
1995 NCSA `httpd 1.3`

1996 Smashing the Stack for Fun & Profit

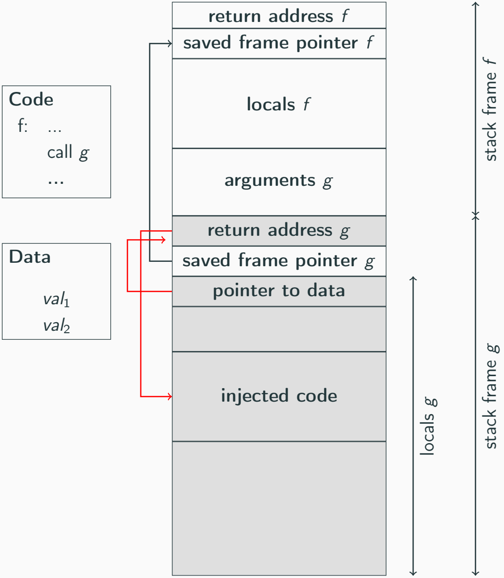
Basic exploitation



Frame pointer overwriting



Indirect pointer overwriting



Example 1

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

int check_authentication(char *password) {
        int auth_flag = 0;
        char password_buffer[16];
        strcpy(password_buffer, password);
        if (strcmp(password_buffer, "kernighan") == 0)
                auth_flag = 1;
        if (strcmp(password_buffer, "ritchie") == 0)
                auth_flag = 1;
        return auth_flag;
}

int main(int argc, char *argv[]) {
        if (argc < 2) { printf("Usage: %s <password>\n", argv[0]); exit(0); }
        if (check_authentication(argv[1])) printf("\nAccess Granted.\n");
        else printf("\nAccess Denied.\n");
}
```

Example 2

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

int check_authentication(char *password) {
        char password_buffer[16]; /* Putting buffers before variables to imped
        int auth_flag = 0;
        strcpy(password_buffer, password);
        if (strcmp(password_buffer, "brillig") == 0)
                auth_flag = 1;
        if (strcmp(password_buffer, "outgrabe") == 0)
                auth_flag = 1;
        return auth_flag;
}

int main(int argc, char *argv[]) {
        if (argc < 2) { printf("Usage: %s <password>\n", argv[0]); exit(0); }
        if (check_authentication(argv[1])) printf("\nAccess Granted.\n");
        else printf("\nAccess Denied.\n");
}
```

Constraints

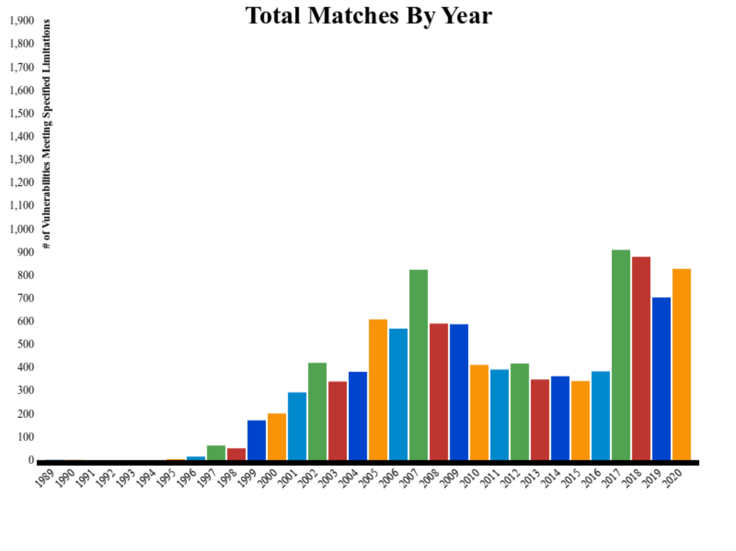
Needs

- Hardware willing to execute data as code
- No null bytes

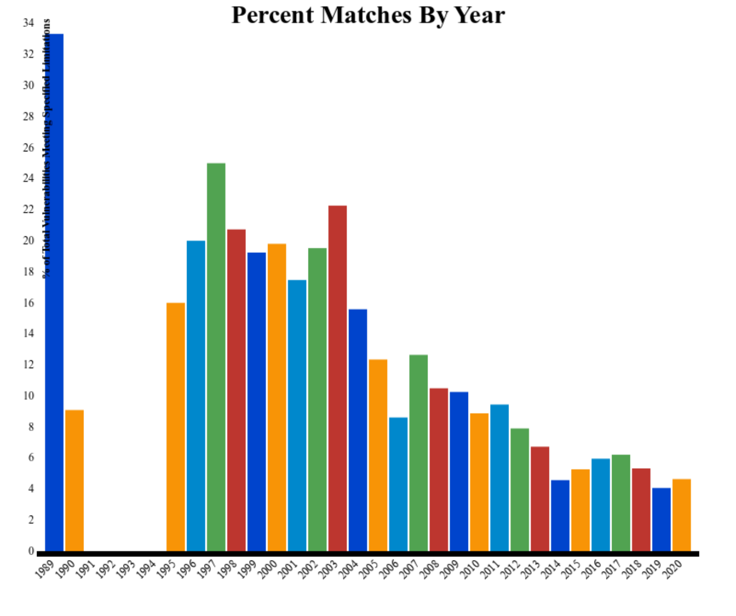
Variants

- Frame pointer corruption
- Causing an exception to execute a specific function pointer

Statistics # (https://nvd.nist.gov/vuln)



Statistics % (`https://nvd.nist.gov/vuln`)



Heap-based buffer overflows

Vulnerability

Heap memory is dynamically allocated at runtime.

Arrays on the heap overflow just as well as those on the stack.

Warning

The heap grows towards higher addresses instead of lower addresses.

This is the opposite of the stack.

Basic exploitation

Overwriting heap-based function pointers located after the buffer

Overwriting virtual function pointer

1998	IE4 Heap overflow
2002	Slapper worm (Linux, OpenSSL)
CVE-2007-1365	OpenBSD 2nd remote exploits in 10 years
CVE-2017-11779	Windows DNS client

Overwriting heap-based function pointers

```
typedef struct _vulnerable_struct
{
     char buff[MAX_LEN];
     int (*cmp)(char*,char*);

} vulnerable;

int is_file_foobar_using_heap(vulnerable* s, char* one, char* two)
{
     strcpy( s->buff, one );
     strcat( s->buff, two );
     return s->cmp(s->buff, "foobar");
}
```

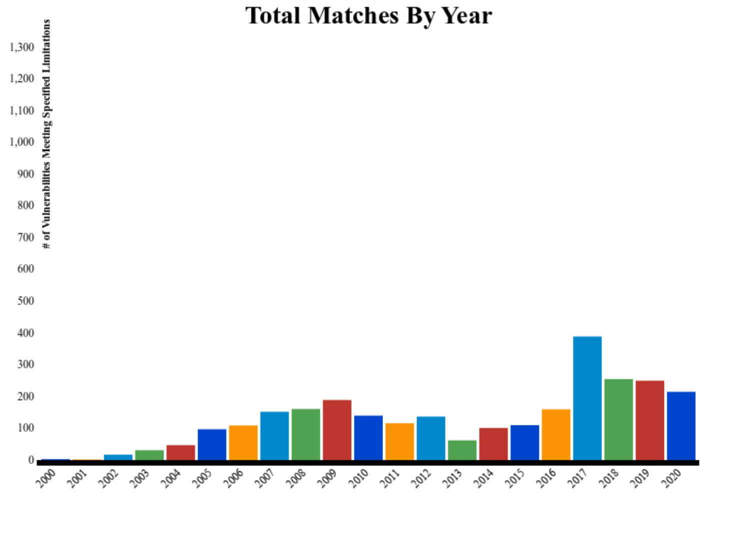
Constraints

- Ability to determine the address of heap
- If string-based, no null-bytes

Variants

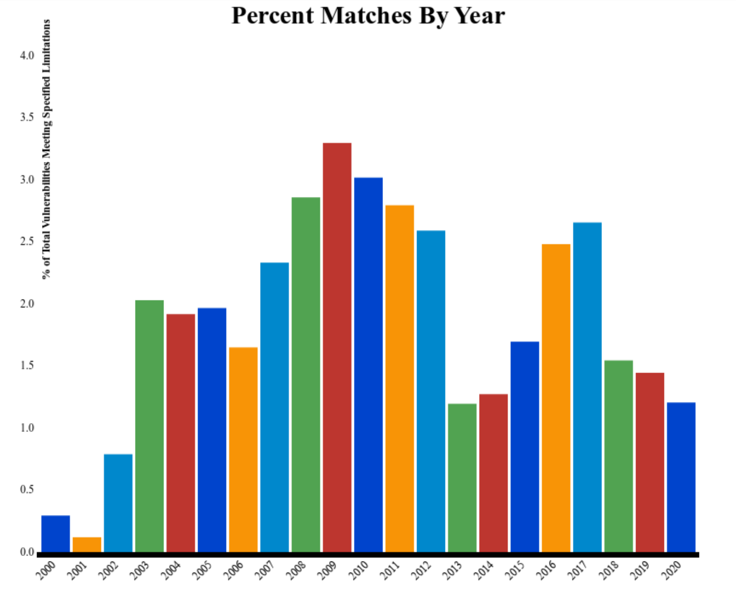
- Corrupt pointers in other (adjacent) data structures
- Corrupt heap metadata

Statistics # (https://nvd.nist.gov/vuln)



Query

Statistics % (`https://nvd.nist.gov/vuln`)



Format strings

About format strings vulnerabilities

They were the ‘spork‘ of exploitation. ASLR? PIE? NX Stack/Heap? No problem, fmt had you covered.

Vulnerability

Format functions are variadic.

```c
int printf(const char *format, ...);
```

How it works

- The format string is copied to the output unless '%' is encountered.
- Then the format specifier will manipulate the output.
- When an argument is required, it is expected to be on the stack.

Caveat

And so ..

If an attacker is able to specify the format string, it is now able to control what the function pops from the stack and can make the program write to arbitrary memory locations.

CVEs

Software	CVE
Zend	2015-8617
latex2rtf	2015-8106
VmWare 8x	2012-3569
WuFTPD (providing remote root since 1994)	2000

Good & Bad

Good ✓

```c
int f (char *user) {
    printf("%s", user);
}
```

Bad ✗

```c
int f (char *user) {
    printf(user);
}
```

Exploitation

Badly formatted format parameters can lead to :

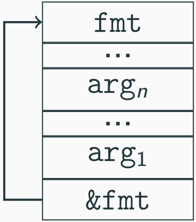
- arbitrary memory read (data leak)
- arbitrary memory write
    - rewriting the `.dtors` section
    - overwriting the Global Offset Table (`.got`)

Example

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

int main(int argc, char *argv[]) {
    char text[1024];
    static int test_val = 65;
    if (argc < 2) {
        printf("Usage: %s <text to print>\n", argv[0]);
        exit(0);
    }
    strcpy(text, argv[1]);
    printf("The right way to print user-controlled input:\n");
    printf("%s", text);
    printf("\nThe wrong way to print user-controlled input:\n");
    printf(text);
    // Debug output
    printf("\n[*] test_val @ 0x%08x = %d 0x%08x\n",
           &test_val, test_val, test_val);
    exit(0);
}
```

Stack situation



Reading from arbitrary addresses

The %s format specifier can be used to read from arbitrary addresses

```
$ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
The right way to print user-controlled input:
AAAA%08x.%08x.%08x.%08x
The wrong way to print user-controlled input:
AAAAffffcbc0.f7ffcfd4.565555c7.41414141
[*] test_val @ 0x56557028 = 65 0x00000041
```

Printing local variable

```
$ ./fmt_vuln $(printf "\x28\x70\x55\x56")%08x.%08x.%08x.%s
The right way to print user-controlled input:
(pUV%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
(pUVffffcbc0.f7ffcfd4.565555c7.A
[*] test_val @ 0x56557028 = 65 0x00000041
```

65 is the ASCII value of 'a'

Writing to arbitrary memory

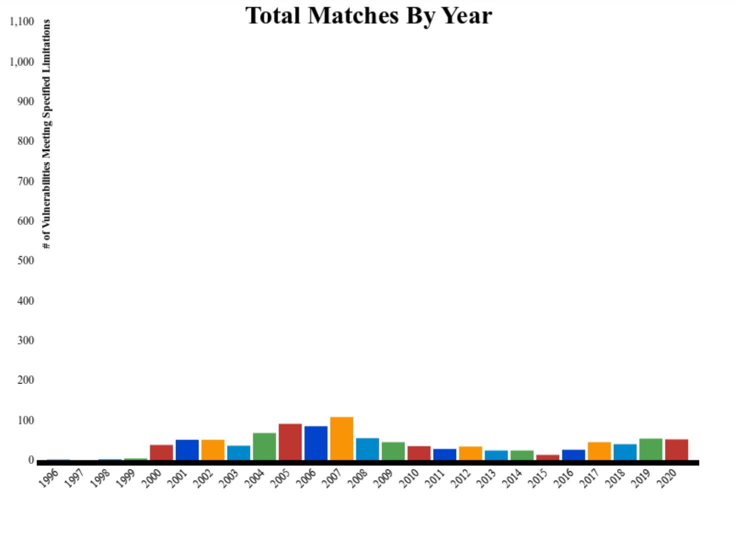
As %s, %n can be used to write to arbitrary addresses.

```
$ ./fmt_vuln $(printf "\x28\x70\x55\x56")%08x.%08x.%08x.%n
The right way to print user-controlled input:
(pUV%08x.%08x.%08x.%n
The wrong way to print user-controlled input:
(pUVffffcbc0.f7ffcfd4.565555c7.
[*] test_val @ 0x56557028 = 31 0x0000001f
```

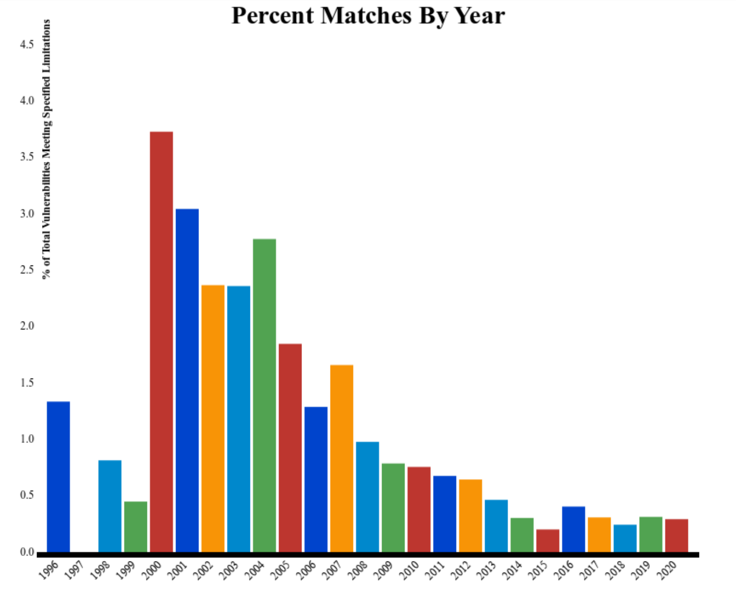
It may be unintentional

- `printf("100% dave")` prints stack entry above saved `eip`
- `printf("%s")` prints bytes pointed to by that stack entry
- `printf("%d %d %d ...")` prints a series of stack entries as integer
- `printf("%08x %08x %08x ...")` same but as hexadecimal values
- `printf("100% no way")` writes 3 to the address pointed to by stack entry

Statistics # (`https://nvd.nist.gov/vuln`)



Statistics % (`https://nvd.nist.gov/vuln`)



Looking back

	Buffer overflow	Format string
public since	$\approx$ 1985	1999
dangerous	1990's	2000
# exploits	thousands	dozens
considered	security threat	programming bug
techniques	evolved & advanced	basic
visibility	sometimes hard	easy

Play (exploitation) games

`https://microcorruption.com`

Questions ?

https://rbonichon.github.io/teaching/2021/asi36/