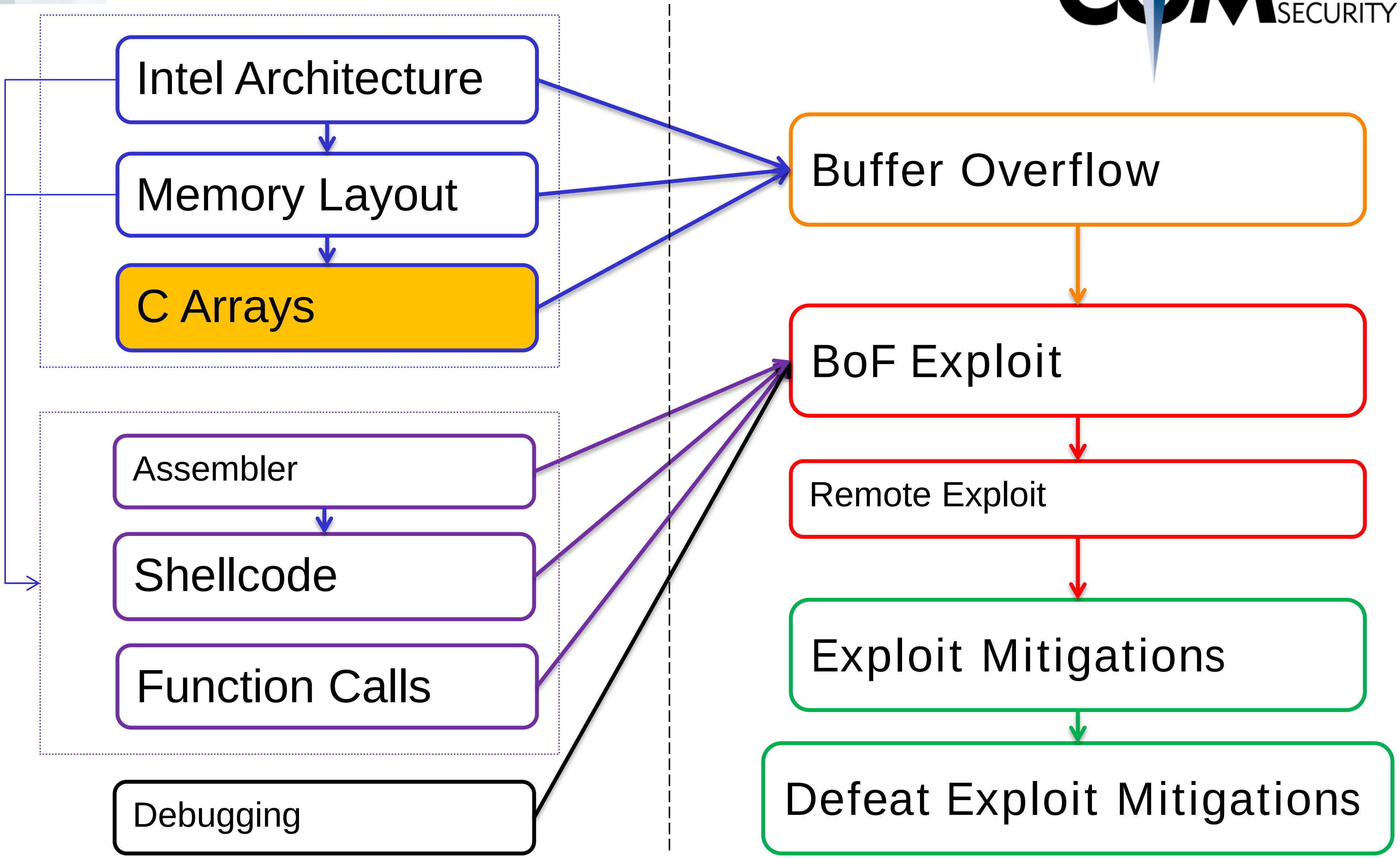
C Arrays and Pointers

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

Content



Intel Architecture

Memory Layout

C Arrays

Assembler

Shellcode

Function Calls

Debugging

Buffer Overflow

BoF Exploit

Remote Exploit

Exploit Mitigations

Defeat Exploit Mitigations

C Arrays & Pointers

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

C Arrays & Pointers

Valid C code:

```
int array[5] = {1, 2, 3, 4, 5};

array[0] = 0;
array[5] = 0;

array[-1] = 0;
array[100] = 0;
```

Valid!

C Arrays & Pointers

Valid C code:

```
int array[5] = {1, 2, 3, 4, 5};

int *a = array;
a += 100;
*a = 0;
```

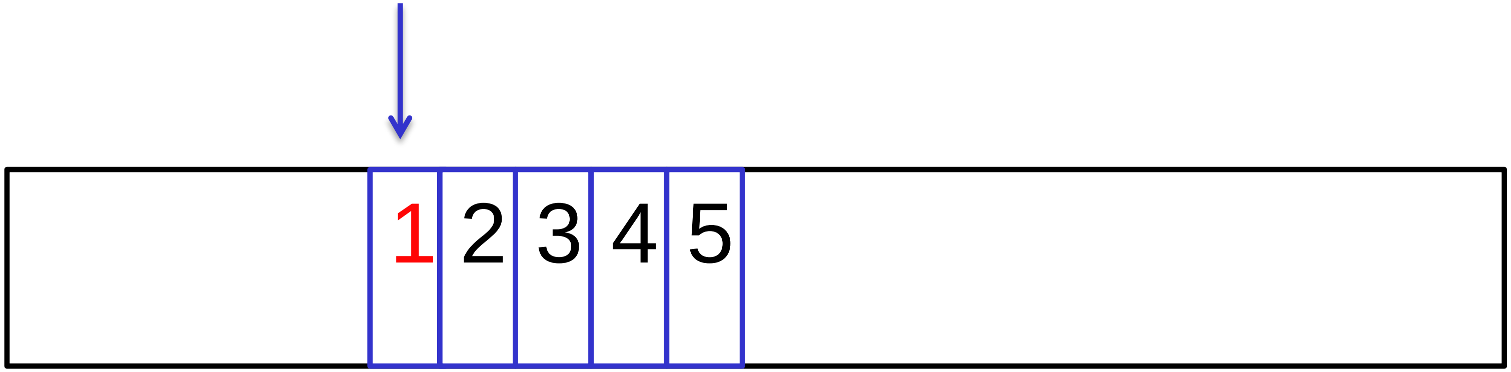
```
array    = a             = 0x1000
array[2] = a + 2 * 4     = 0x1008


(int is 32 bit = 4 bytes)
```

C Arrays & Pointers

Valid C code:

```
int array[5] = {1, 2, 3, 4, 5};

int *a = array;
```

*array = *a = 1



C Arrays & Pointers

Other c code:

```
int a = 42;
int *b = &a;

printf(“%i”, a);     // 42
printf(“%i”, *b);    // 42

b++;

printf(“%i”, *b);    // ??
```

C Arrays & Pointers

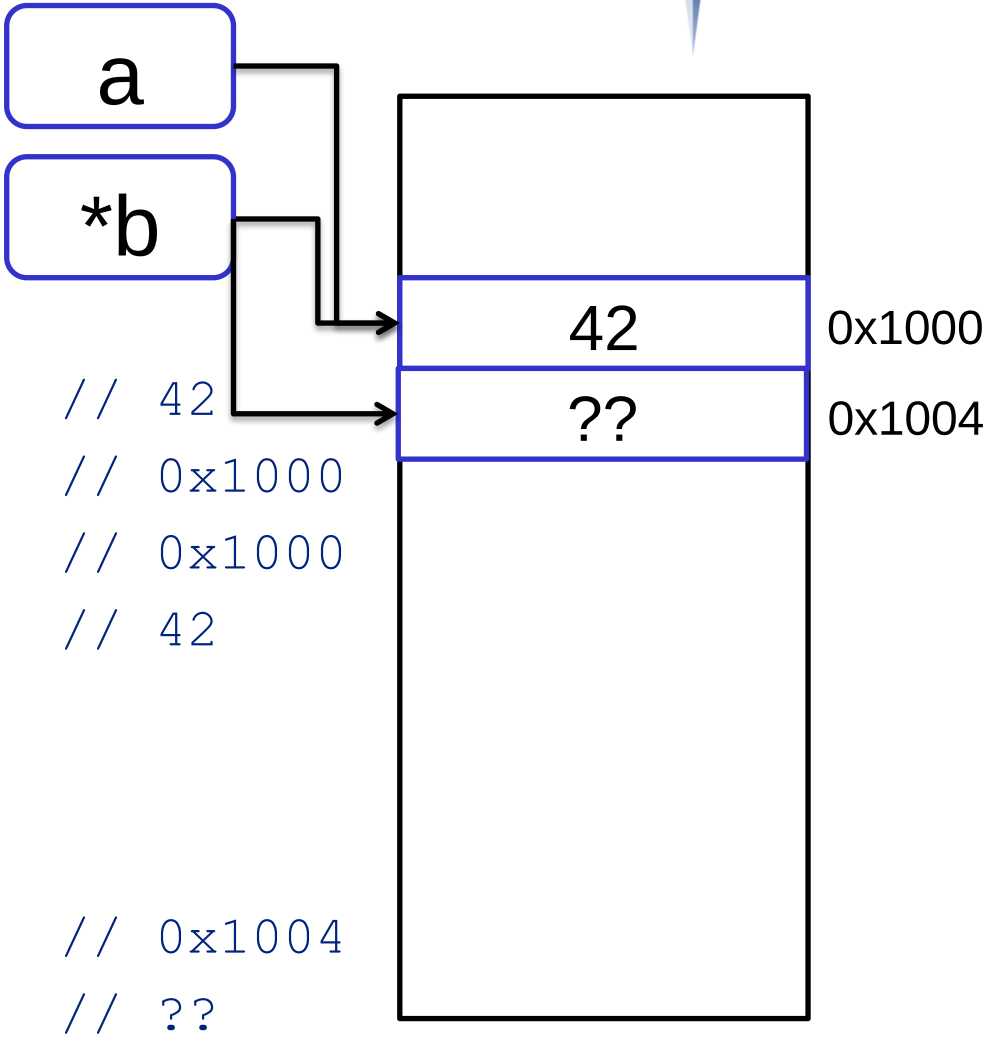
Other c code:

```
int a = 42;
int *b = &a;

printf(“%i”, a);      // 42
printf(“%i”, &a);     // 0x1000
printf(“%i”, b);      // 0x1000
printf(“%i”, *b);     // 42

b++;

printf(“%i”, b);      // 0x1004
printf(“%i”, *b);     // ??
```

C Arrays & Pointers

Other c code:

```
    int a = 42;
    int *b = &a;

    printf("%i", a);     // 42
    printf("%i", &a);    // 0x1000
    printf("%i", b);     // 0x1000
    printf("%i", *b);    // 42

    b++;

    printf("%i", b);     // 0x1004
    printf("%i", *b);    // ??
```



strcpy()

Compass Security Schweiz AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

Exploitation Basics

What is a common vulnerability?

```
strcpy(destination, source);
strcpy(d, “Hallo”);
```

Exploitation Basics

What is a common vulnerability?

```
strcpy(destination, source);
strcpy(d, “Hallo”);
```

How much does strcpy() actually copy?

- Until source “ends”
- Where is the end?
- 0 byte \x00

```
“Hallo\x00”
```

Exploitation Basics

strcpy() does not care about destination size

At all

```
char destination[8];
char source[16] = “1234567890123456”

strcpy(destination, source);
```

Exploitation Basics

strcpy() does not care about destination size

At all, because:

```
char destination[8];
char *d = &destination;
char source[16] = “1234567890123456”

strcpy(d, source);
```

An memory curruption example

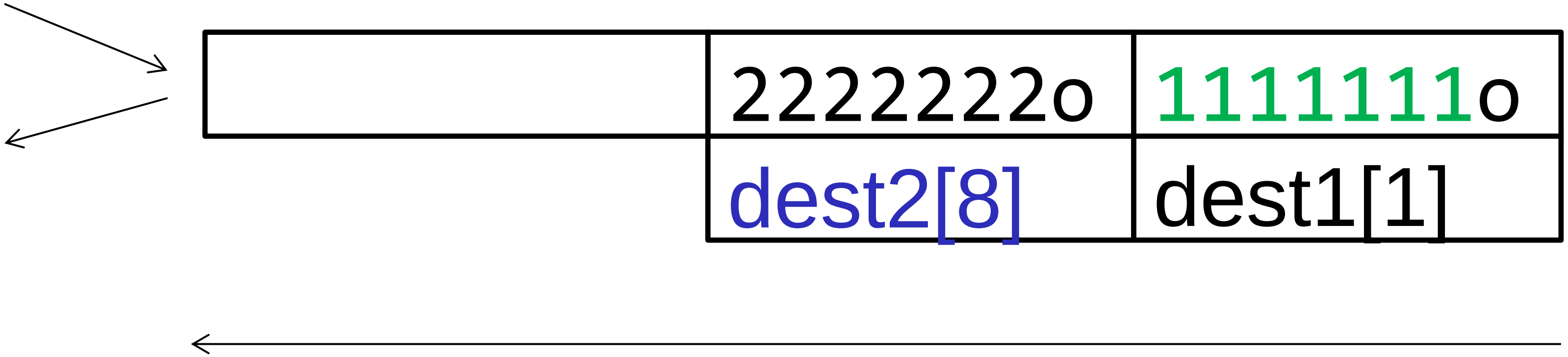
Exploitation Basics

```
#include <stdio.h>

void main(int argc, char **argv) {
  char dest1[8] = "1111111";
  char dest2[8] = "2222222";

  strcpy(dest2, argv[1]);

  printf("%p Dest1 : %s\n", dest1, dest1);
  printf("%p Dest2 : %s\n", dest2, dest2);
}
```

Exploitation Basics

Normal behaviour:

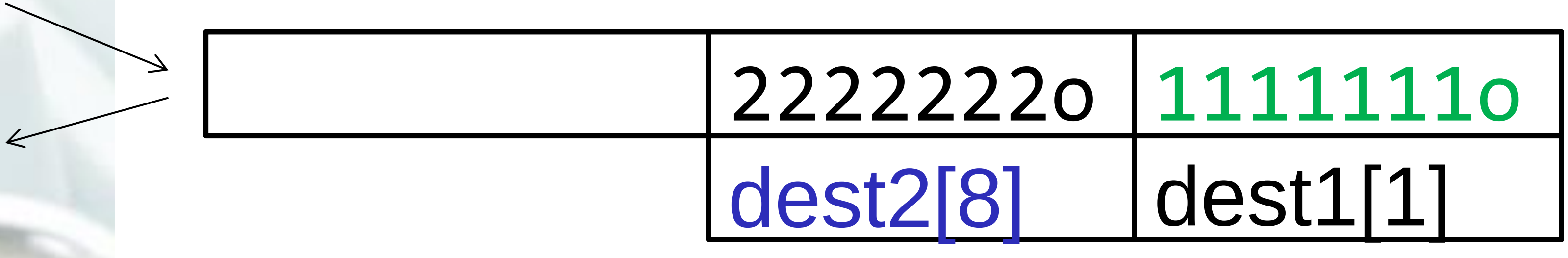
```
    char dest1[8] = "1111111";
    char dest2[8] = "2222222";

    strcpy(dest2, argv[1]);
```



	2222222o	1111111o
	dest2[8]	dest1[1]

Exploitation Basics

Normal behaviour: Init



2222222o

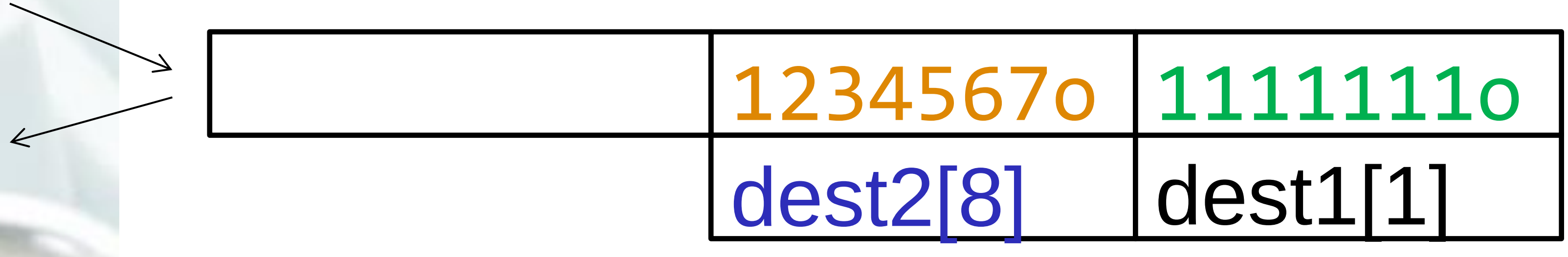
1111111o

dest2[8]

dest1[1]

Exploitation Basics

Normal behaviour: After strcpy()

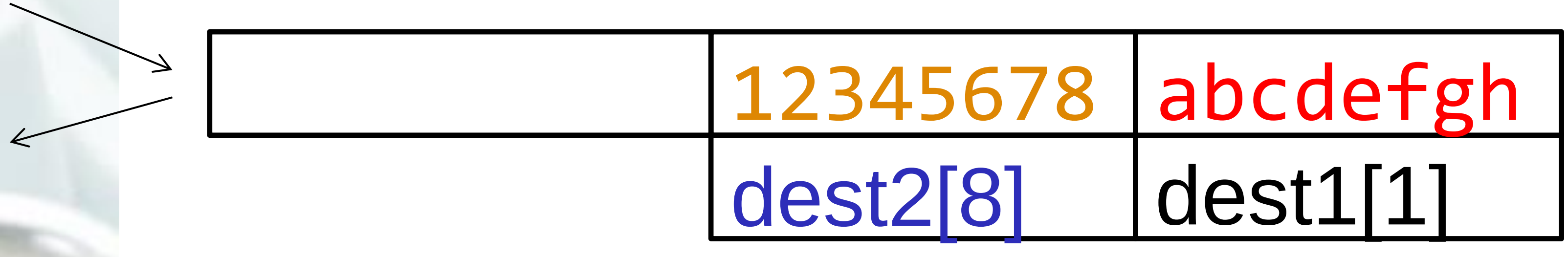


	1234567o	1111111o
	dest2[8]	dest1[1]

```
$ ./strcpy 1234567
0xbfa6c438 Dest1 : 1111111
0xbfa6c430 Dest2 : 1234567
```

Exploitation Basics

Abnormal behaviour: After strcpy()



	12345678	abcdefgh
	dest2[8]	dest1[1]

```
$ ./strcpy 12345678abcdefgh
0xbfbe7588 Dest1 : abcdefgh
0xbfbe7580 Dest2 : 12345678abcdefgh
```

Exploitation Basics

Some random x64 architecture facts:

The stack should stay **8-byte aligned** at all times

An n-byte item should start at an **address divisible by n**

- E.g. 64 bit number: 8 bytes, can be at 0x00, 0x08, 0x10, 0x18, …

%rsp points to the lowest **occupied** stack location

- not the next one to use!

Conclusion

Exploitation Basics

Recap:

- C does not care about buffer boundaries
- strcpy() does not care about end of buffer (only 0-byte)
- One buffer can overflow into another buffer
- Local variables/buffers are adjectant to each other
- Pointer can point to any memory address