Introduction to Software Vulnerability Exploitation (2017)

Ricardo Narvaja – Daniel Kazimirow

Based on gera’s dojo – ReCon 2010

Computer basics

The Memory: Relativistic meaning

0x434f5245 (int, big endian)

0x45524f43 (int, little endian)

1000011 1001111 1010010 1000101

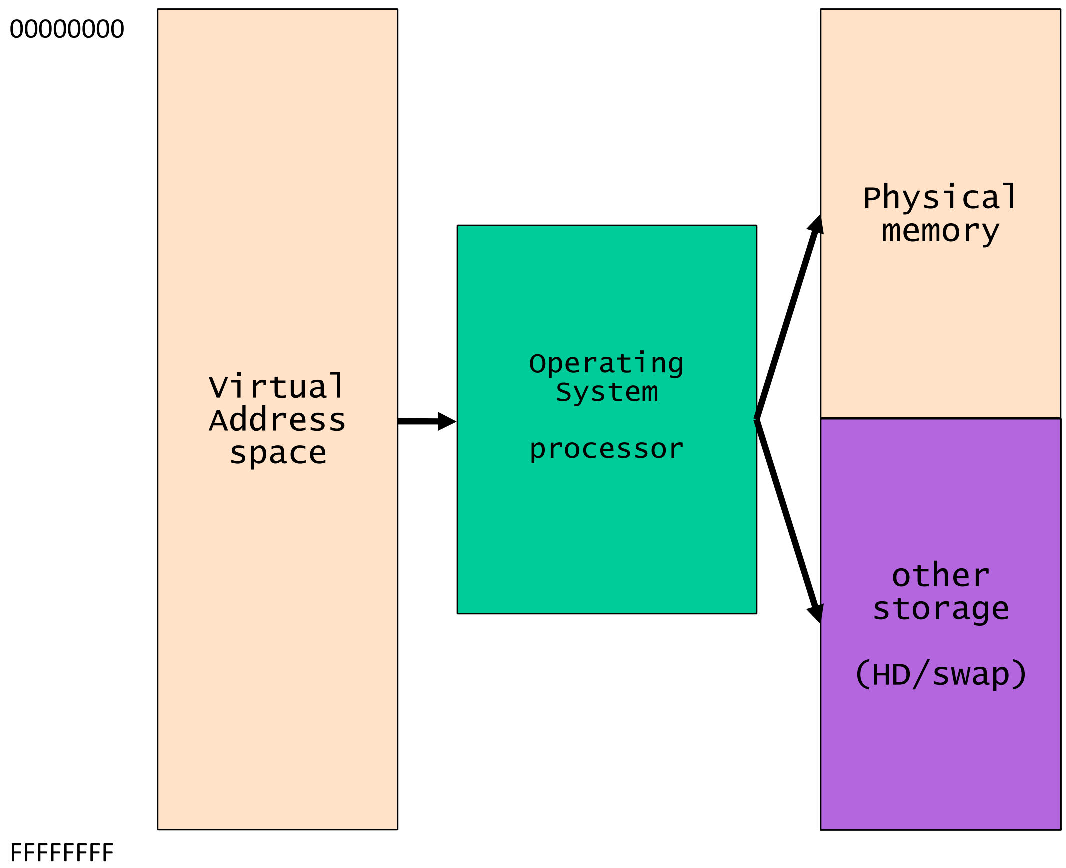
43 4f 52 45

C O R E (or even E R O C)

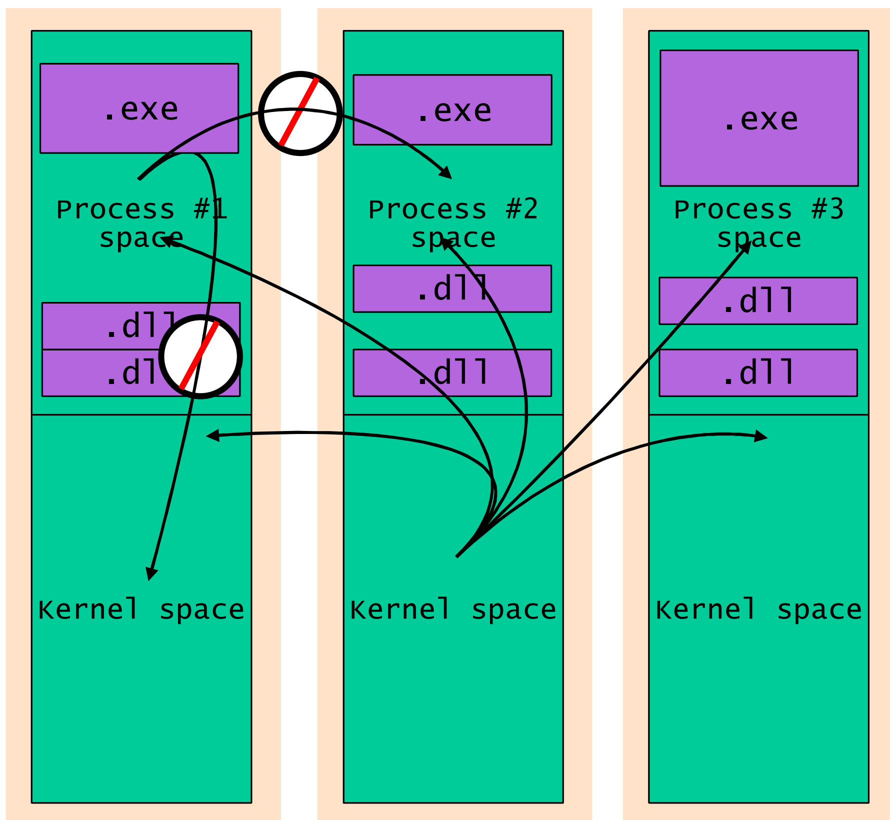
The Memory: Addressing

byte 0	08	01	00	BD
byte 4	31	AB	11	10
		.		
		.		
byte 100	4A	21	65	89
byte 104	4A	21	65	89
byte 108	4A	21	65	89
byte 10c	4A	21	65	89
		.		
		.		
byte n	2D	3F	6A	2D
byte n+4	45	24	10	76
byte n+8	25	46	79	80

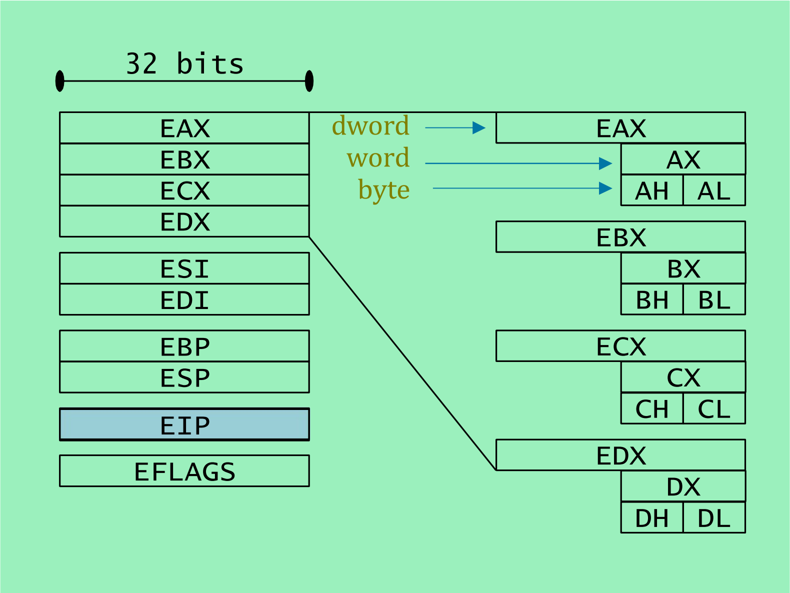
The Memory: Addressing



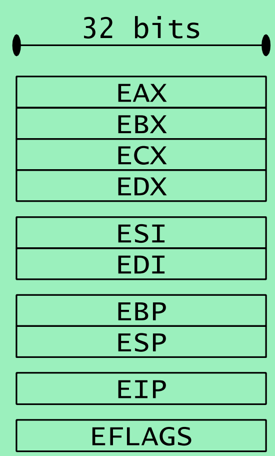
The Memory: Memory map of a windows process



Microprocessor: Registers – IA32 / AMD32

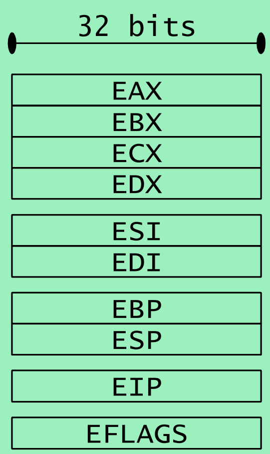


Microprocessor: Assembly Simple Operations




```
mov     ecx, 14
mov     ecx, eax
mov     eax, [eax]
mov     [eax], edx
mov     ebx, [eax+100]
mov     [eax+ebx*8+100], edx
```

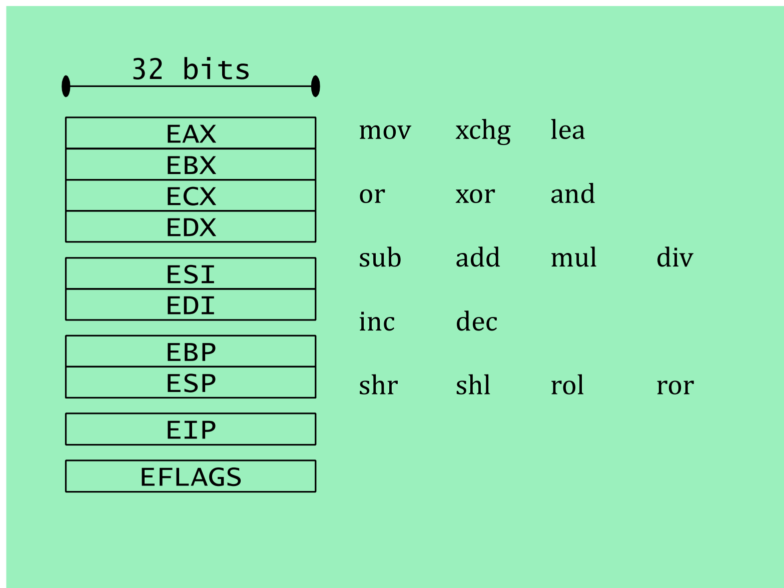
Microprocessor: Assembly Simple Operations




```
add     ecx, 14
add     ecx, eax
add     eax, [eax]
add     [eax], edx
add     ebx, [100+eax]
add     [100+eax+ebx*8], edx
```

CORE SECURITY

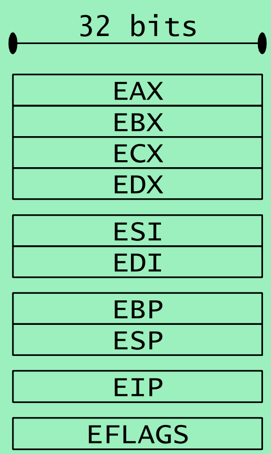
Microprocessor: Assembly Simple Operations



CORE SECURITY

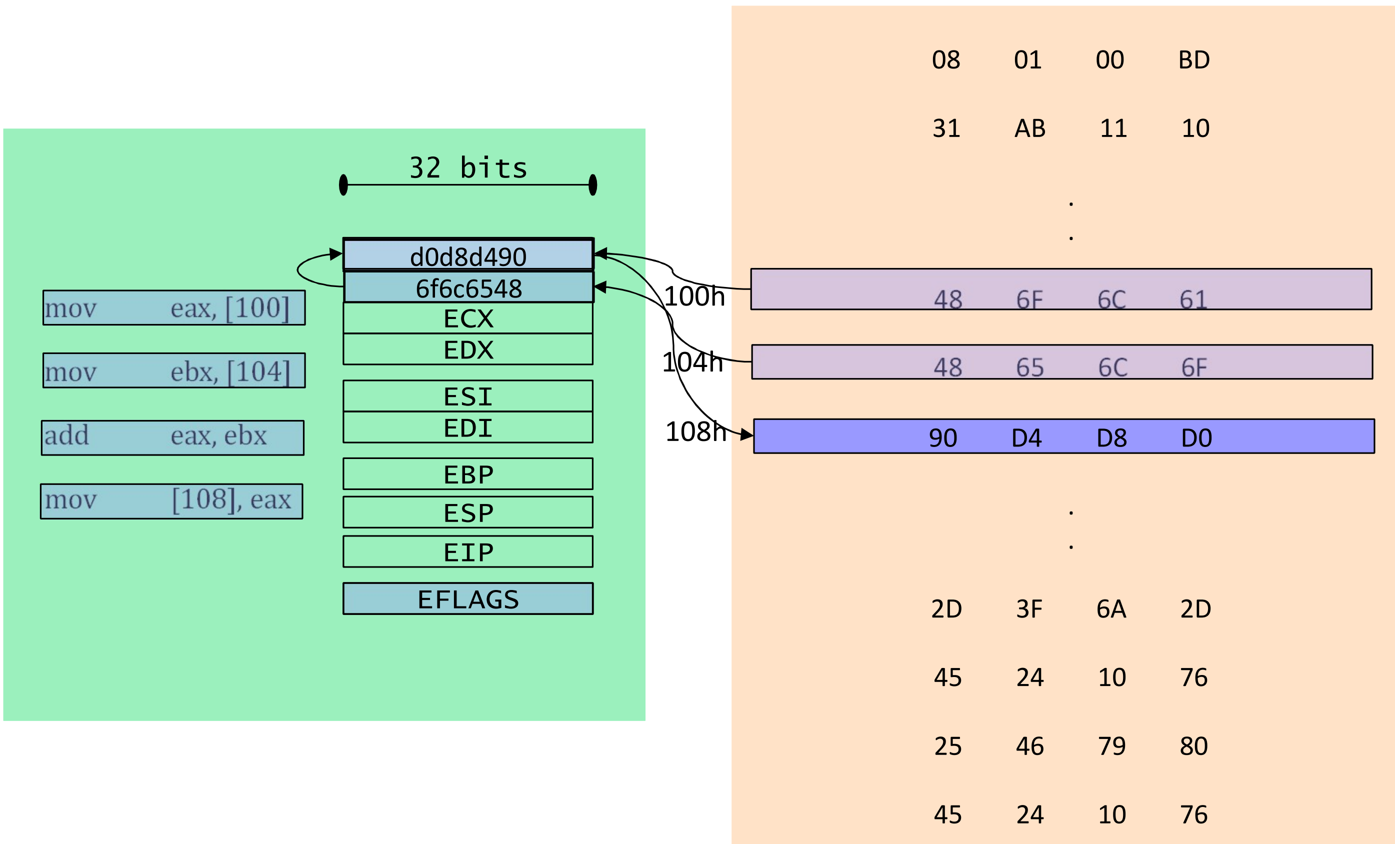
Microprocessor: Assembly Simple Operations

lea doesn't access memory

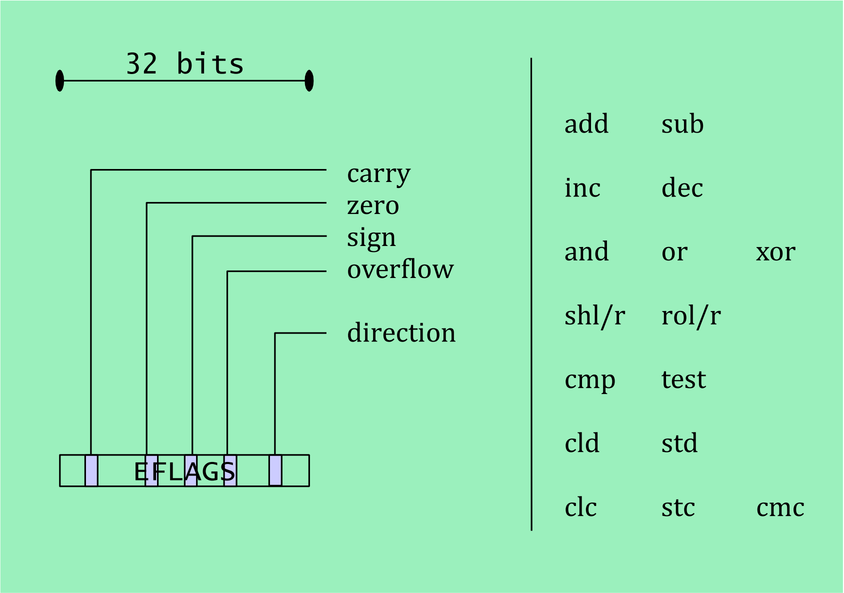



```
lea     eax, [eax+eax*4]
lea     ecx, [eax+edx+1000]
lea     esp, [edx-1000]
lea     ebp, [edi+esp*8+44]
lea     esi, [esi]
lea     edi, [1234]
```

Microprocessor: Assembly Simple Operations



Microprocessor: Assembly Flags



Microprocessor: Assembly Branching Operations

Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	0F 80
JNO	Jump if not overflow		OF = 0	71	0F 81
JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE JZ	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	0F 85
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	0F 82
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	0F 83
JBE JNA	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1	76	0F 86
JA JNBE	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0	77	0F 87

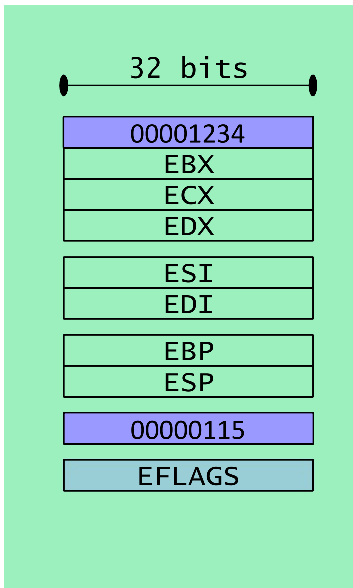
CORE SECURITY

Microprocessor: Assembly Branching Operations

Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JL JNGE	Jump if less Jump if not greater or equal	signed	SF <> OF	7C	0F 8C
JGE JNL	Jump if greater or equal Jump if not less	signed	SF = OF	7D	0F 8D
JLE JNG	Jump if less or equal Jump if not greater	signed	ZF = 1 or SF <> OF	7E	0F 8E
JG JNLE	Jump if greater Jump if not less or equal	signed	ZF = 0 and SF = OF	7F	0F 8F
JP JPE	Jump if parity Jump if parity even		PF = 1	7A	0F 8A
JNP JPO	Jump if not parity Jump if parity odd		PF = 0	7B	0F 8B
JCXZ JECXZ	Jump if %CX register is 0 Jump if %ECX register is 0		%CX = 0 %ECX = 0	E3	

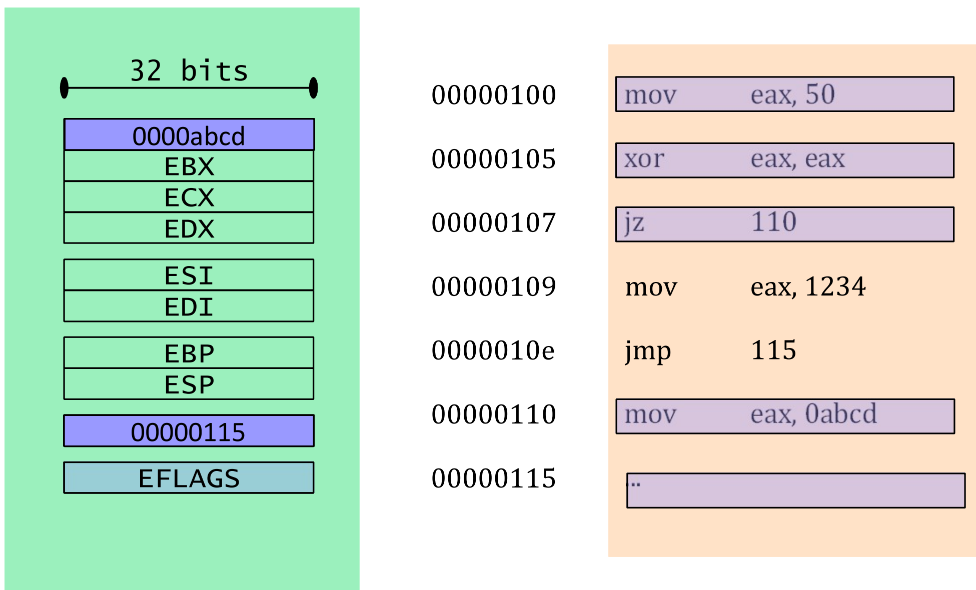
CORE SECURITY

Microprocessor: Assembly Branching Operations

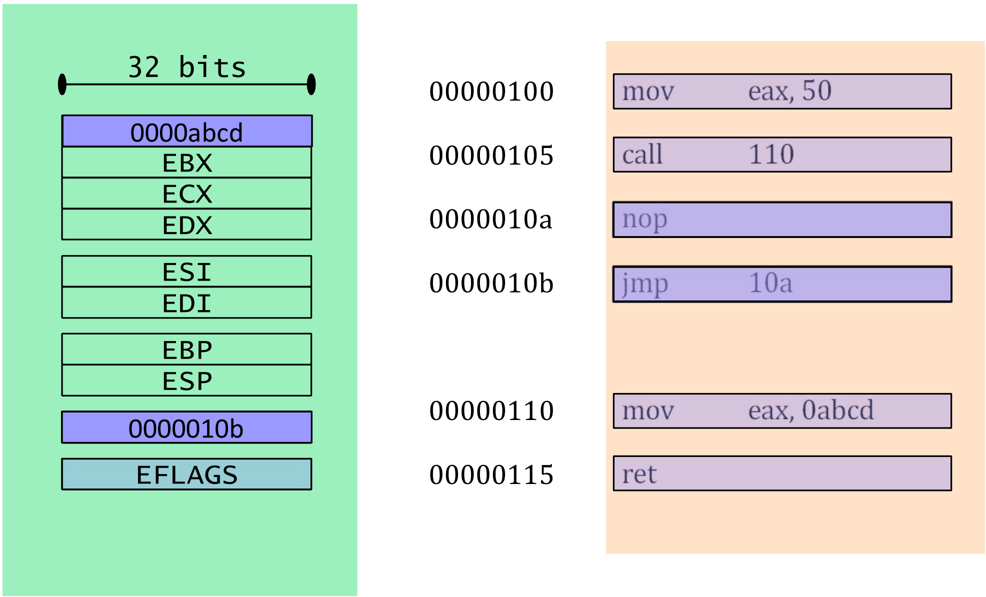


00000100	mov	eax, 50
00000105	or	eax, eax
00000107	jz	110
00000109	mov	eax, 1234
0000010e	jmp	115
00000110	mov	eax, 0abcd
00000115	...	

Microprocessor: Assembly Branching Operations



Microprocessor: Assembly Branching Operations



Microprocessor: Signed vs Unsigned

C code

```
#define MAX_SIZE 80

int main(int argc, char **argv)
{
        unsigned int len;
        char buf[MAX_SIZE ];
        ...
        len = get_len();

        if (len > MAX_SIZE )
                exit();

        memcpy(buf, data, len);
}
```

Assembly

```
main:

        ...

        call get_len()
        mov        [len],eax

        cmp        eax, 50
        jbe        not_big
        call       exit
not_big:
        ...
        call       memcpy
```

Microprocessor: Signed vs Unsigned

C code

```c
#define MAX_SIZE 80

int main(int argc, char **argv)
{
        int len;
        char buf[MAX_SIZE ];
        ...
        len = get_len();

        if (len > MAX_SIZE )
                exit();

        memcpy(buf, data, len);
}
```

Assembly

```
main:

        ...

        call get_len()
        mov     [len],eax

        cmp     eax, 50
        jle     not_big
        call    exit
not_big:
        ...
        call    memcpy
```

memcpy

Visual Studio 6.0

Copies characters between buffers.

void *memcpy(void ***_dest_, const void ***_src_**, size_t** _count_ **);**

size_t unsigned __int64 or unsigned integer, depending on the target platform)	Result of sizeof operator.

Microprocessor: Are 32 bits enough?

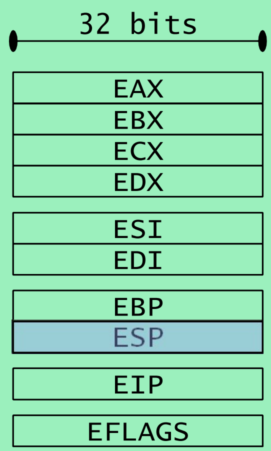
C code

```c
int main(int argc, char **argv)
{
        unsigned len;
        int *list;
        ...
        len = get_len();

        list = malloc(len * 4);

        memcpy(list, data, len *  4);
}
```

Assembly

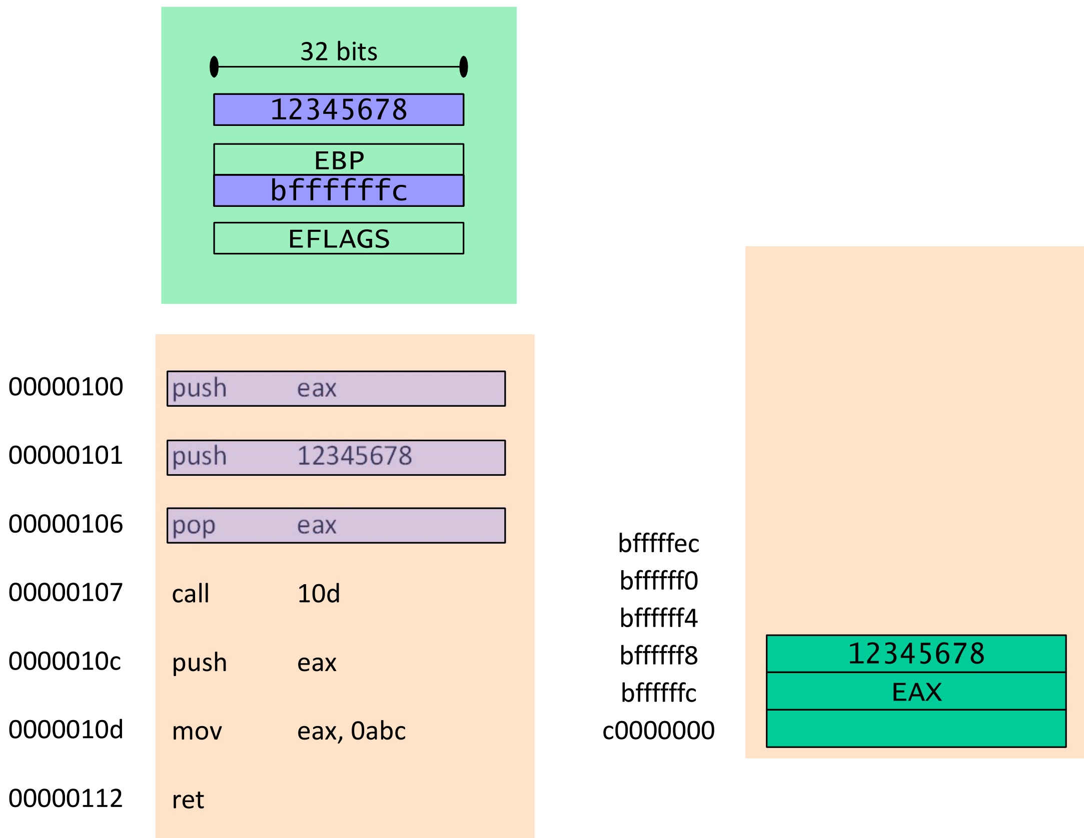
```
main:

        ...

        call get_len()
        mov         [len],eax

        lea         eax, [eax*4]
        ...
        call        malloc

        ...
        call        memcpy
```

Microprocessor: Stack Operations



push	pop
pushf	popf
pusha	popa
call	ret

Microprocessor: Assembly Stack Operations



CORE SECURITY

Microprocessor: C Calling Convention

C code

```
void _start() {
    ...
    main(argc, argv, envp);
    ...
    exit(2);
}
```

Assembly

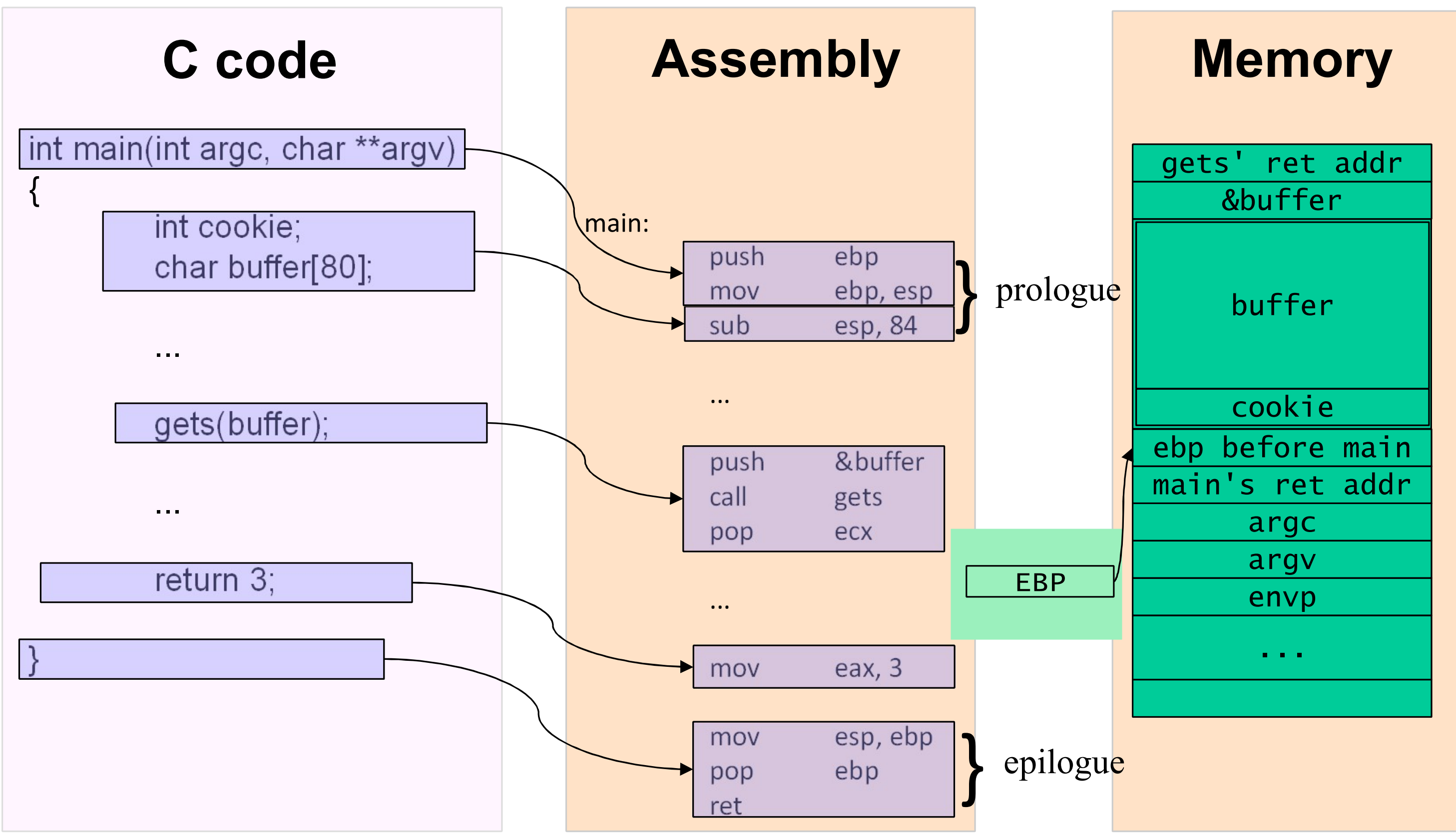
```
_start:
        ...
        push    envp
        push    argv
        push    argc
        call    main
        add     esp, c
        ...
        push    2
        call    exit
```

Memory

main's ret addr
argc
argv
envp
...

CORE SECURITY

Microprocessor: C Calling Convention



Microprocessor: C Calling Convention

C code

```
void _start() {
    ...

    main(argc, argv, envp);

    ...

    exit(2);
}
```

Assembly

```
_start:
        ...
        push    envp
        push    argv
        push    argc
        call    main
        add     esp, c
        ...
        push    2
        call    exit
```

Memory

argc
exit's ret addr
2
...

CORE SECURITY

Understanding and exploiting the bugs

Understanding the bugs: Buffer Overflow

stack1.c:

```
int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);
        if (cookie == 0x41424344)
           printf("you win!\n");
}
```

buf	80 bytes
cookie	4 bytes
EBP	4 bytes
main's return addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes

CORE SECURITY

Understanding the bugs: Buffer Overflow

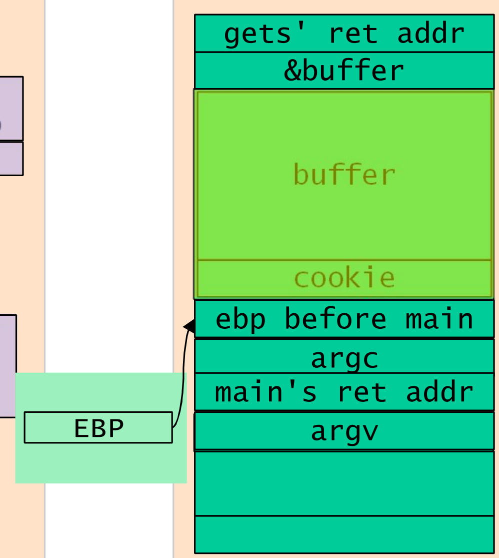
C code

```
int main(int argc, char **argv)
{
        int cookie;
        char buffer[80];

        ...

        gets(buffer);

        ...

        return 3;

}
```

Assembly

```
main:
      push    ebp
      mov     ebp, esp
      sub     esp, 54

      ...

      push    &buffer
      call    gets
      pop     ecx

      ...

      mov     eax, 3

      mov     esp, ebp
      pop     ebp
      ret
```

Memory



Understanding the bugs: Buffer Overflow

stack2.c:

```
int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);
        if (cookie == 0x01020305)
           printf(“you win!\n”);
}
```

buf	80 bytes
cookie	4 bytes
EBP	4 bytes
main's return addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes

CORE SECURITY

Understanding the bugs: Buffer Overflow

stack3.c:

```
int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);
        if (cookie == 0x01020005)
           printf(“you win!\n”);
}
```

buf	80 bytes
cookie	4 bytes
EBP	4 bytes
main's return addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes

CORE SECURITY

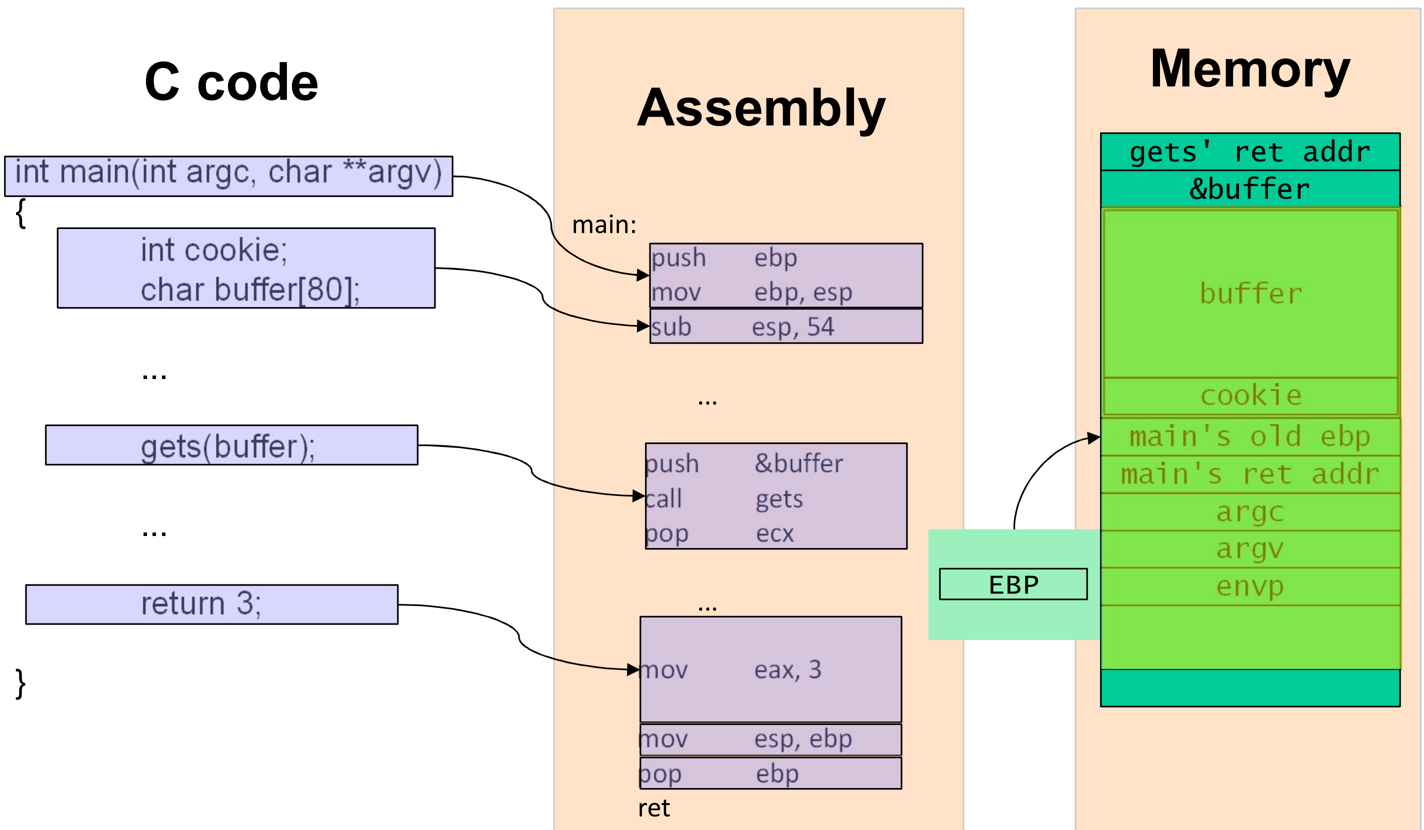
Understanding the bugs: Buffer Overflow

stack4.c:

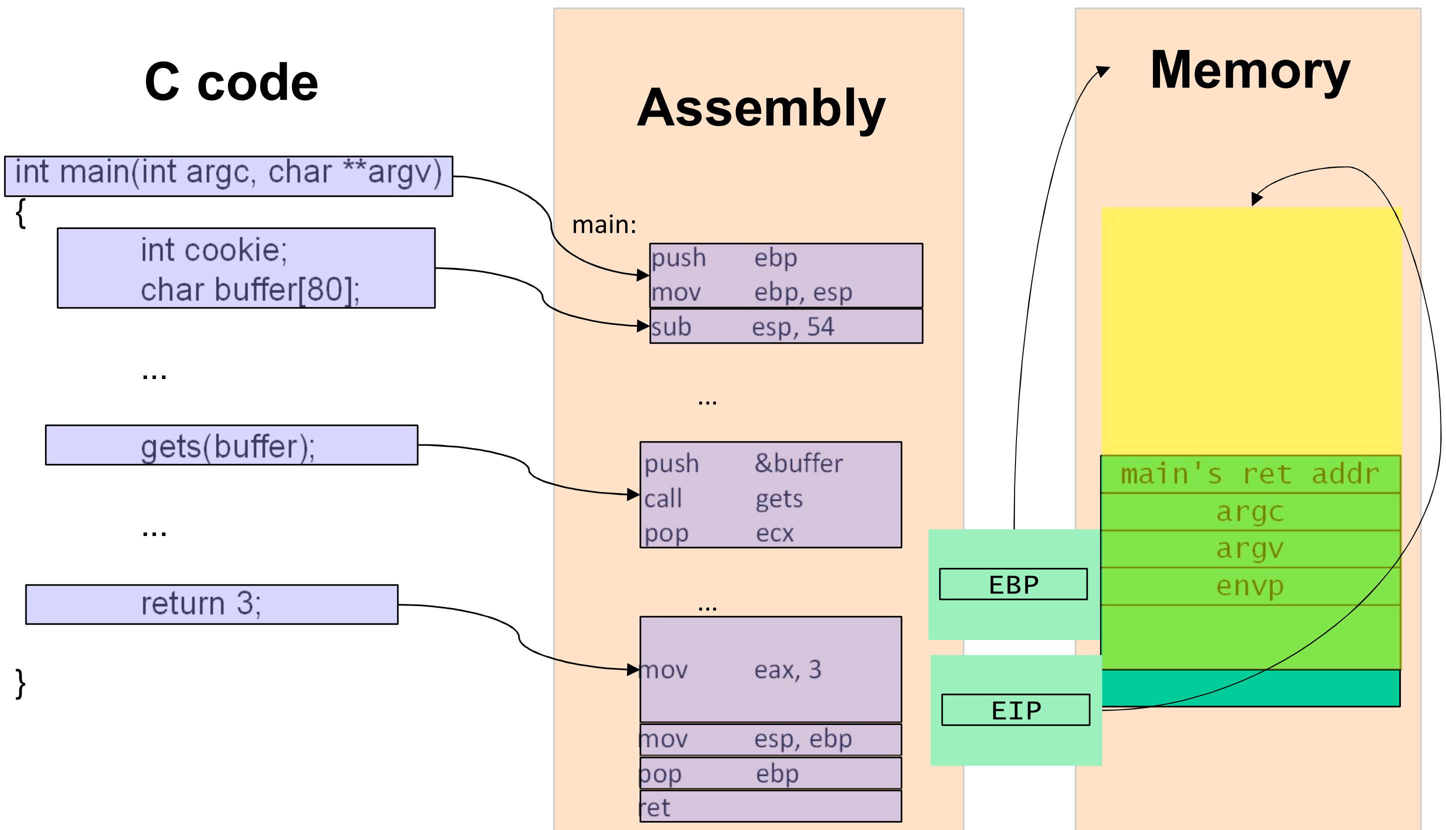
```
int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);
        if (cookie == 0x000d0a00)
          printf(“You win!\n”);
}
```

buf	80 bytes
cookie	4 bytes
EBP	4 bytes
main's return addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes

Understanding the bugs: Buffer Overflow



Understanding the bugs: Buffer Overflow

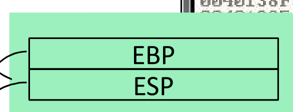


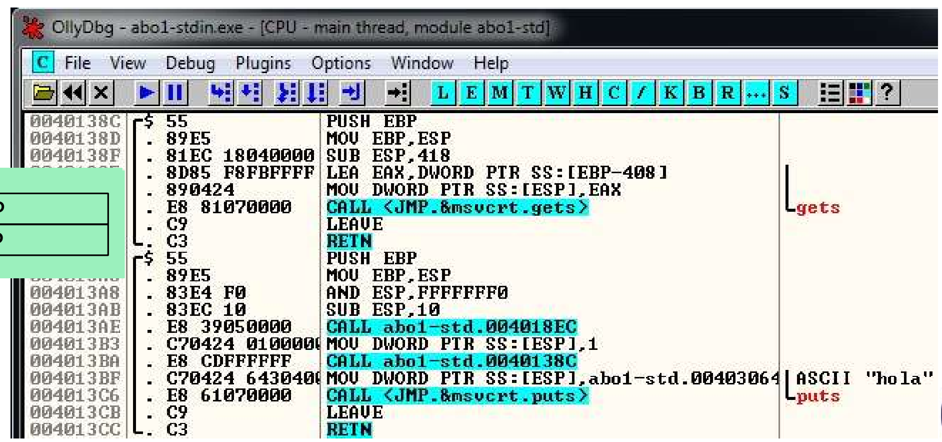
Understanding the bugs: Buffer Overflow

abo1.c:

```c
void f(int nada) {
        char buf[1024];
        gets(buf);
}

int main(int argc,char **argv) {
        f(1);
        printf("hola\n");
}
```

gets's return addr	4 bytes
compiler vars	24 bytes
buf	1024 bytes
EBP	4 bytes
main's return addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes





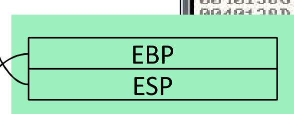
CORE SECURITY

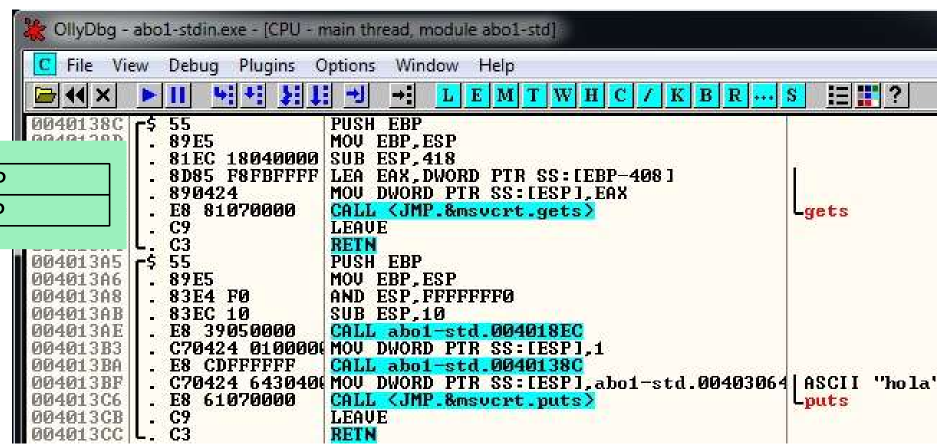
Understanding the bugs: Buffer Overflow

abo1.c:

```
void f(int nada) {
        char buf[1024];
        gets(buf);
}

int main(int argc,char **argv) {
        f(1);
        printf("hola\n");
}
```

gets's return addr	4 bytes
compiler vars	24 bytes
buf	1024 bytes
EBP	4 bytes
main's return addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes



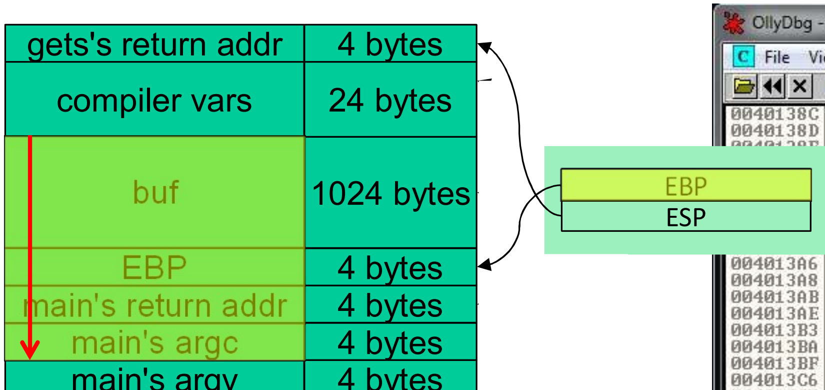


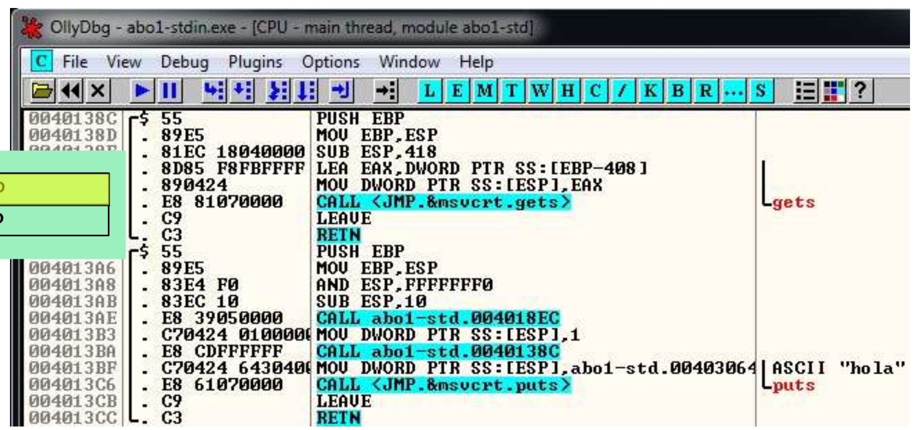
CORE SECURITY

Understanding the bugs: Buffer Overflow

abo1.c:

```
void f(int nada) {
        char buf[1024];
        gets(buf);
}

int main(int argc,char **argv) {
        f(1);
        printf("hola\n");
}
```

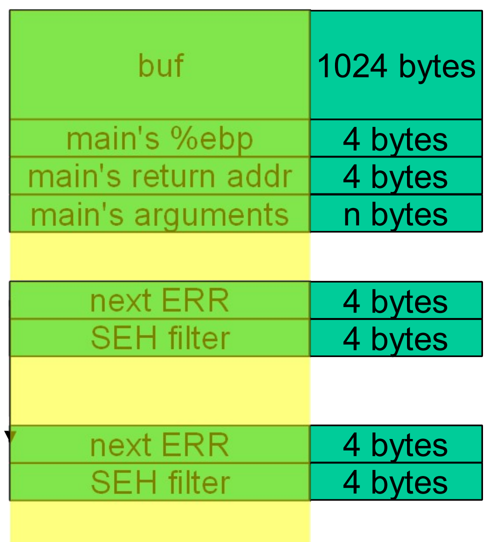




Exploiting the bugs: Buffer overflow – Exploiting SEH

abo2.c:

```
int main(int argc,char **argv) {
        char buf[1024];
        gets(buf);
        exit(1);
}
```



Exploiting the bugs: Complex Buffer Overflow

abo3.c:

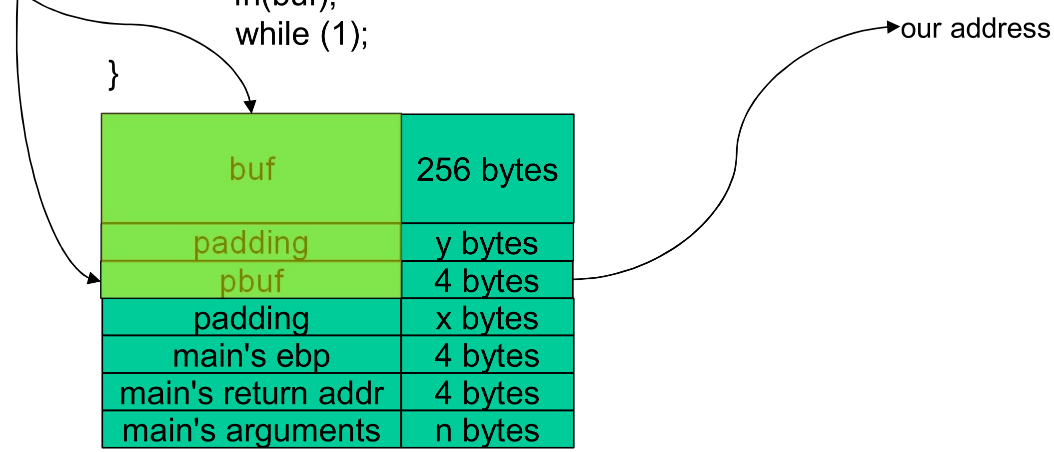
```
int main(int argc,char **argv) {
        extern system,puts;
        void (*fn)(char*)=(void(*)(char*))&system;
        char buf[256];

        fn=(void(*)(char*))&puts;
        gets(buf);
        fn(argc[2]);
        exit(1);
}
```

buf	256 bytes
fn's ebp	4 bytes
fn's return addr	4 bytes
fn's "function" arg	4 bytes

Exploiting the bugs: Complex Buffer Overflow

abo4.c (part of it):

```
void (*fn)(char*)=(void(*)(char*))&system;
int main(int argc,char **argv) {
        char *pbuf=malloc(100);
        char buf[256];

        gets(buf);
        gets(pbuf);
        fn(buf);
        while (1);
}
```



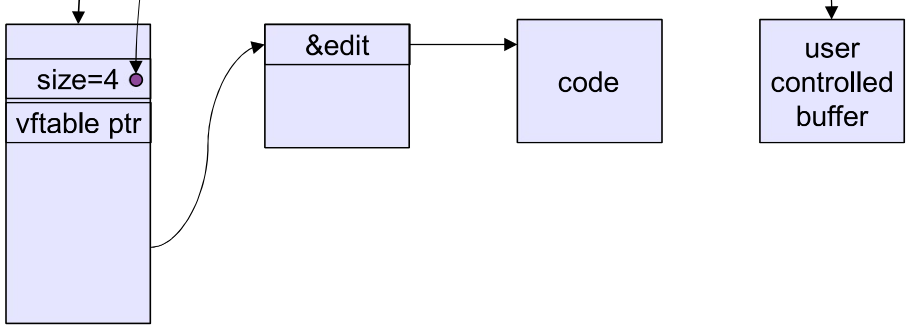
write-anything-anywhere primitive

Use-After-Free

Exploiting the bugs: Use-After-Free

heap0.c

```
int main(int argc, char **argv) {
  char buf[80];
  fgets(buf, sizeof(buf), stdin);

 contact *c = new contact();
 delete c
 char *pbuf = (char *)malloc(4);
  c->edit(1, "someone");
}
```

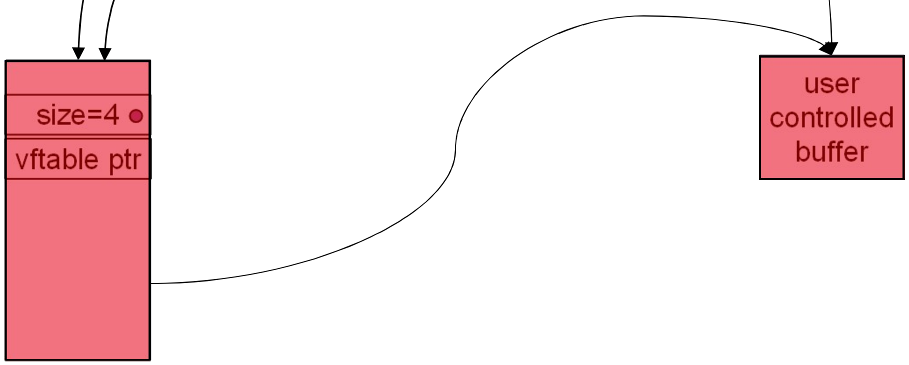


CORE SECURITY

Exploiting the bugs: Use-After-Free

heap0.c

```
int main(int argc, char **argv) {
  char buf[80];
  fgets(buf, sizeof(buf), stdin);

 contact *c = new contact();
 delete c
 char *pbuf = (char *)malloc(4);
  c->edit(1, "someone");
}
```



Understanding the bugs: Use-After-Free – Hands on!

```
/* .. who's there? */

using namespace std;

class contact {
    public:
        virtual void edit(unsigned int contact, string
name);
};

void contact::edit(unsigned int contact, string name) {
    cout << "editing " << name << endl;
}

int main(int argc, char **argv) {
  char buf[256];
  fgets(buf, 256, stdin);
  contact *c = new contact();
  delete c;
  char *p1 = (char *)malloc(sizeof(contact));
  fgets(p1, 4, stdin);
  c->edit(1, "someone");
}
```

CORE SECURITY

Signed vs Unsigned (or “computers vs numbers”)

Understanding the bugs: Signed vs Unsigned

- A register/dword can be **positive** or **negative**

- The **sign** is determined by the **highest bit**

- A 32 bits number can represents:
  - Unsigned number: 0 ~ 4294967295
  - Signed number: -2147483648 ~ 2147483647

- Range:
  - 0x80000000 … 0 … 0x7FFFFFFF

Understanding the bugs: Signed vs Unsigned

- Ex:
  - v = 3 → 0x00000003
  - v = -3 → 0xFFFFFFFD
- Negating a number (3 → -3):
  - v = 3 → NOT (0x00000003) → 0xFFFFFFFC + 1
  - v = -3 → 0xFFFFFFFD
- Comparing unsigned numbers:
  - If (0x00000001 < 0x7FFFFFFF) … ?
  - If (0xFFFFFFFF < 0x00000001) … ?
  - If (0x7FFFFFFF < 0x80000000) … ?

Understanding the bugs: Signed vs Unsigned

- Comparing signed numbers:
  - If (0x00000001 < 0x7FFFFFFF) … ?
  - If (0xFFFFFFFF < 0x00000001) … ?
  - If (0x7FFFFFFF < 0x80000000) … ?

- Comparing numbers in ASM:
  - EAX = 0x00000001
  - EBX = 0xFFFFFFFF

  - "cmp eax,ebx"
    - "jb 0x80808080" (JUMP is **BELOW**) … ?
    - "jl 0x80808080" (JUMP is **LESS**) … ?

CORE SECURITY

Integer Overflow

Understanding the bugs: Integer Overflow

- It's produced when the **result** of an **arithmetic operation** is **too large** to be represented within the available storage space.

- It's produced by operations like:
  - Addition, substraction, multiplication and division.

- Unsigned 32 bit numbers:
  - Range: 0 ~ 4294967295
  - 4294967295 + 1 = ? → 0 … **INTEGER OVERFLOW !**
  - 0 – 1 = ? → 4294967295 … **INTEGER UNDERFLOW !**

CORE SECURITY

Protection mechanisms

Windows Protection Mechanisms

- Canary (Cookies)
- Data Execute Prevention (bit NX)
- ASLR
- Windows SEH Protections
- Heap Protections

Protections: Stack – Stackguard

example.c

```
int main() {
        char buf[80];
        gets(buf);
    }
```

buf	80 bytes
ebp before main	4 bytes
canary	4 bytes
main's ret addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes

CORE SECURITY

Protections: Propolice and /GS

example.c

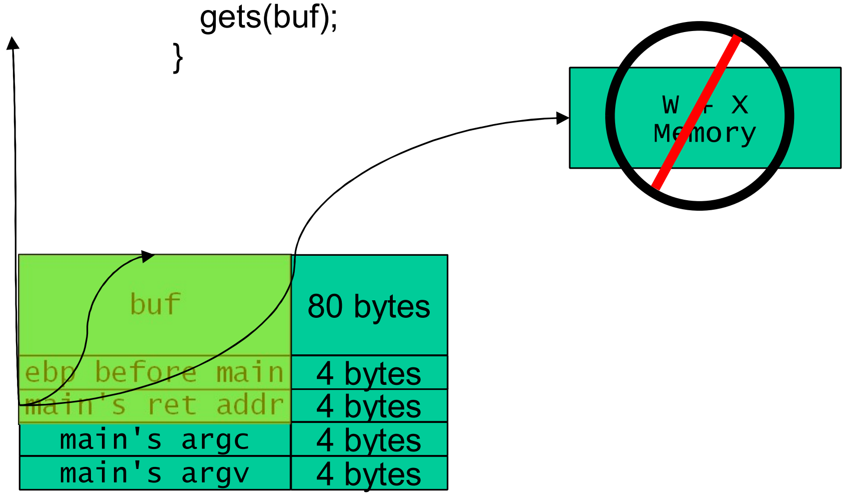
```
int main() {
        char buf[80];
        gets(buf);
}
```

cookie	4 bytes
buf	80 bytes
canary	4 bytes
esi before main	4 bytes
edi before main	4 bytes
ebp before main	4 bytes
main's ret addr	4 bytes
main's argc	4 bytes
main's argv	4 bytes

Protections: W^X - DEP

example.c

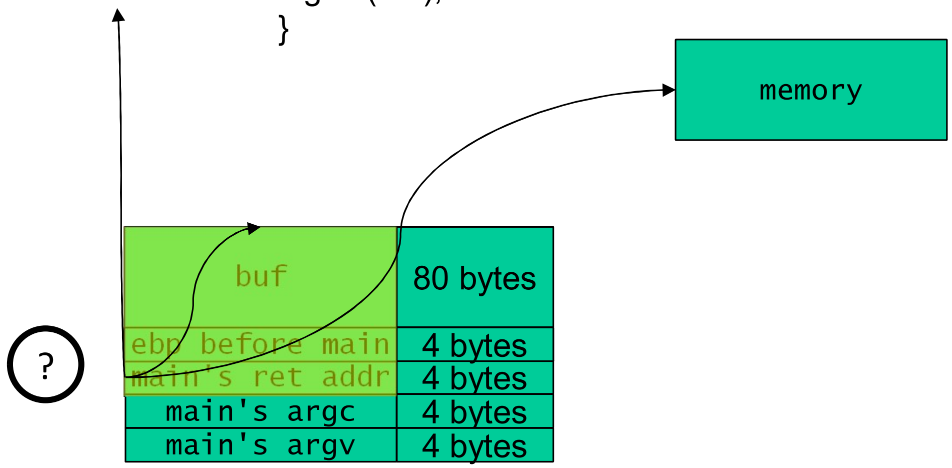
```
int main() {
        char buf[80];
        gets(buf);
}
```



Protections: ASLR

example.c

```
int main() {
        char buf[80];
        gets(buf);
    }
```



Protections: Windows SEH protections

- SE Handler pointing to code
  - Must not be in stack
  - DEP – NX (Heap is not an option)

- SAFE SEH
  - Handlers white list (only modules compiled with “/SafeSEH”)

- SEHOP
  - The last “SEH handler” has to point to →
    “ntdll!FinalExceptionHandler” (ASLR is the problem !)

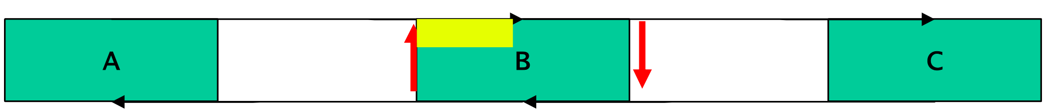
CORE SECURITY

Protections: Heap protections

example.c

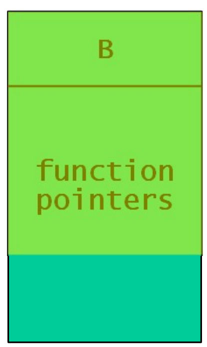
```
int main() {
        ...

        free( B );

}
```

- safe unlink
- cookies
- pointer encoding
- direct mmap



Protections: Heap protections

- next block overwrite
- other techniques



Protections: Hardware DEP

- This protection takes advantage of the NX bit (No Execute page)
- NX marks parts of the memory (data regions) as non executable. The processor will then refuse to execute any code residing in these areas of memory.

Protections: Hardware DEP

- The DEP's behavior in Windows XP /2003 can be changed via a boot.ini parameter.

- Starting from Windows Vista, DEP state can be changed by using the **bcedit** command.

```
bcdedit.exe /set nx OptIn
bcdedit.exe /set nx OptOut
bcdedit.exe /set nx AlwaysOn
bcdedit.exe /set nx AlwaysOff
```

Protections: Hardware DEP – The APIs

The most important API added is SetProcessDEPPolicy, which sets the DEP policy for the running process.

SetProcessDEPPolicy function

Changes data execution prevention (DEP) and DEP-ATL thunk emulation settings for a 32-bit process.

Syntax

C++

```cpp
BOOL WINAPI SetProcessDEPPolicy(
  _In_  DWORD dwFlags
);
```

Parameters

dwFlags [in]

A **DWORD** that can be one or more of the following values.

Value	Meaning
0	If the DEP system policy is OptIn or OptOut and DEP is enabled for the process, setting *dwFlags* to 0 disables DEP for the process.
PROCESS_DEP_ENABLE 0x00000001	Enables DEP permanently on the current process. After DEP has been enabled for the process by setting **PROCESS_DEP_ENABLE**, it cannot be disabled for the life of the process.
[illegible]	Disables DEP-ATL thunk emulation for the

Bypassing DEP: ROP techniques – Gadget

- When hardware DEP is enabled, you cannot just jump to your shellcode
- Gadget definition

> In his original paper, Hovav Shacham used the term "gadget" when referring to higher-level macros/code snippets. Nowadays, the term "gadget" is often used to refer to a sequence of instructions, ending with a ret (which is in fact just a subset of the original definition of a "gadget"). It's important to understand this subtlety, but at the same time I'm sure you will forgive me when I use "gadget" in this tutorial to refer to a set of instructions ending with a RET.

CORE SECURITY

Bypassing DEP: ROP techniques – Gadget example

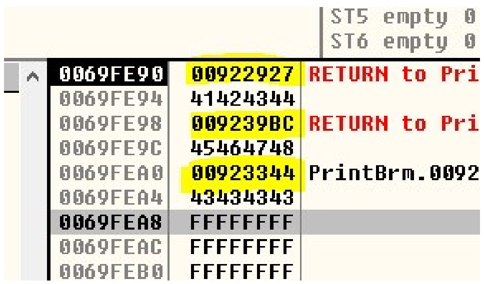
- Example gadget to set ECX to an arbitrary value:

00922927	59	POP ECX
00922928	C3	RETN
00922929	CC	INT3
0092292A	CC	INT3

- Return address must point to the POP ECX/RET gadget; next item on the stack must be the value that will be loaded into ECX.
- In this example, ECX will take the value 0x41424344. After that, execution will continue with the next gadget at 0x9239bc.

0069FE90	00922927	RETURN to PrintBrm.0092
0069FE94	41424344	
0069FE98	009239BC	RETURN to PrintBrm.0092
0069FE9C	45464748	
0069FEA0	EEEEEEEE	
0069FEA4	FFFFFFFF	
0069FEA8	FFFFFFFF	

– VirtualAlloc / VirtualProtect



- For bypassing DEP, the most common technique is building a call to ***VirtualAlloc*** (to allocate a new memory region with executable permissions) or ***VirtualProtect*** (to give executable permissions to an existing memory region) using ROP.
- Our gadgets chain will set the right values for the arguments and finally jump to one of the APIs mentioned above, ensuring to return to our “new” executable code.

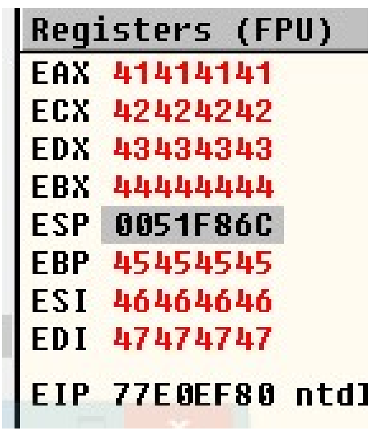
CORE SECURITY

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET

- The techniques used to provide the correct arguments for an API to allow the execution of our code are numerous and depend exclusively on the gadgets found in the application code.

- The most popular one is the PUSHAD-RET-RET technique, because this gagdet allows to easily setup the arguments in the stack, and it’s usually easy to found it in binary code.

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET



- We can use Ollydbg to manually build a ROP chain.
- When the analyzed program crashes, a handy technique is to change the register values to 0x41414141, 0x42424242, etc, **except for ESP**.

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET

60	PUSHAD
C3	RETN
CC	INT3
CC	INT3

- Assemble a PUSHAD - RET gadget in the next instruction to execute.
- Trace till RET by pressing F7.
- The stack will be modified placing our register values in the right position.

0051F84C	47474747
0051F850	46464646
0051F854	45454545
0051F858	0051F86C
0051F85C	44444444
0051F860	43434343
0051F864	42424242
0051F868	41414141
0051F86C	00000000

CORE SECURITY

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET

- Manually changing ESI to point to ***VirtualAlloc***, EDI to a RET instruction and tracing the PUSHAD-RET-RET sequence we can see in the stack the desired arguments for this API.

```
0051F838  45454545  CALL to VirtualAlloc
0051F83C  0051F850  Address = 0051F850
0051F840  44444444  Size = 44444444 (1145324612.)
0051F844  43434343  AllocationType = PAGE_NOACCESS|PAGE_RE
0051F848  42424242  Protect = PAGE_READONLY|PAGE_EXECUTE_R
0051F84C  41414141
```

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET

VirtualAlloc function

Reserves or commits a region of pages in the virtual address space
this function is automatically initialized to zero, unless **MEM_RESE**

To allocate memory in the address space of another process, use t

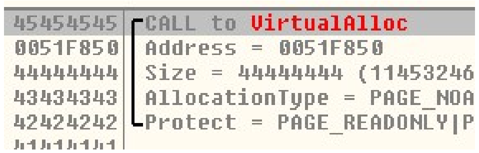
Syntax

C++

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_  LPVOID lpAddress,
  _In_      SIZE_T dwSize,
  _In_      DWORD flAllocationType,
  _In_      DWORD flProtect
);
```

These are the arguments needed for ***VirtualAlloc*** to enable execution of our code buffer. We need to put the right value in the right register, with a chain of gadgets, and finally jump to a PUSHAD–RET gadget.

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET



EBP (45454545) = Return address

EBX (44444444) = Size

EDX (43434343) = Allocation type

ECX (42424242) = Protect

ESI (46464646) = VirtualAlloc

EDI (47474747) = RET

CORE SECURITY

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET

```
45454545  CALL to VirtualAlloc
0051F850  Address = 0051F850
44444444  Size = 44444444 (11453246
43434343  AllocationType = PAGE_NOA
42424242  Protect = PAGE_READONLY|P
```

Setting “Address” argument is not trivial without harcoding, but with the PUSHAD RET-RET technique, ESP will automatically end up in the position of the “Address” argument.

Bypassing DEP: ROP techniques – Using PUSHAD - RET - RET

The correct initial values of the registers previous to PUSHAD-RET-RET must be the following:

EBP = JMP ESP

EBX = Size (0x1)

EDX = 0x1000 (Allocation type: MEM_COMMIT)

ECX = 0x40 (Protect: PAGE_EXECUTE_READWRITE)

ESI = VirtualAlloc

EDI = RET

Thank you