Exploiting Stack Buffer Overflows

TECHNISCHE UNIVERSITÄT DARMSTADT

Learning how blackhats smash the stack for fun and profit so we can prevent it

EC SPRIDE
EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN

Andreas Follner

Graduated earlier this year from the UAC Technikum Wien in Vienna with a Master‘s in IT Security

Works as a research assistant at EC-SPRIDE focussing on buffer overflows

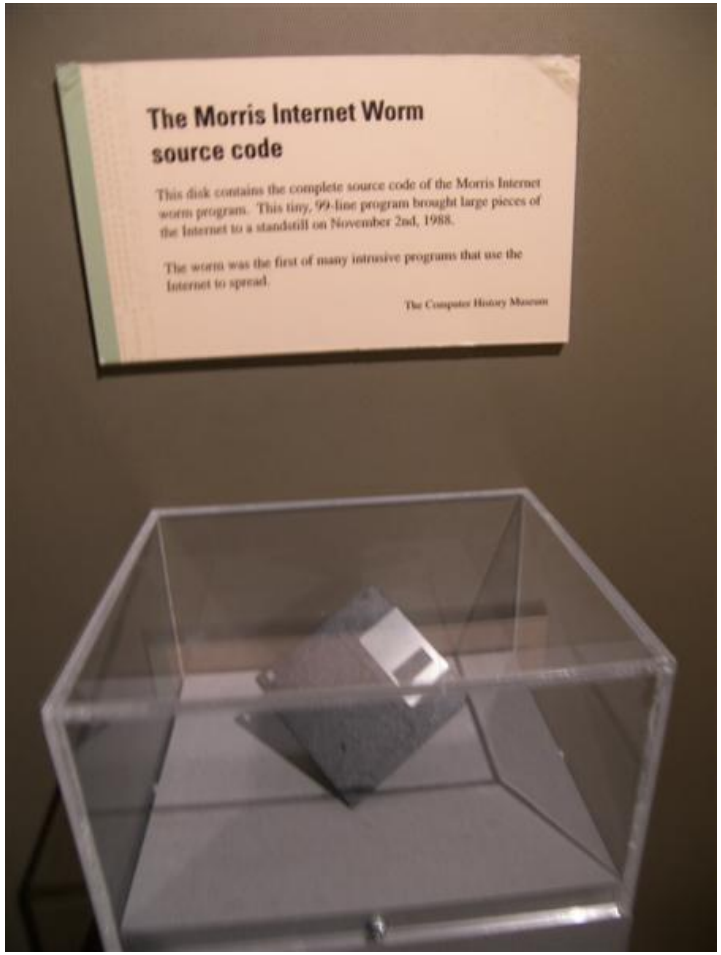
Introduction

- Why exploits?
- What is an exploit?
- What are the dangers of exploits?
- How technical will this presentation be?
- How do exploits work?

Why exploits?

- Because they‘re still a huge threat!

- Worms which exploited buffer overflow vulnerabilities:
  - Morris (1988)
  - SQL Slammer (2003)
  - Conficker (2008)
  - Stuxnet (2010)
  - ... and many more



What is an exploit?

"Typically, a very small program that when utilized causes a software vulnerability to be triggered and leveraged by the attacker."

- James C. Foster

What are the dangers of exploits?

- So far mostly
  - Financial damage
  - Loss of reputation
- In the future?
  - Everything‘s controlled by a computer and connected to the internet (TVs, cars, facilities, …)

RSA
The Security Division of EMC

Contact | Support | Login | Content Library Search

Home > Programs

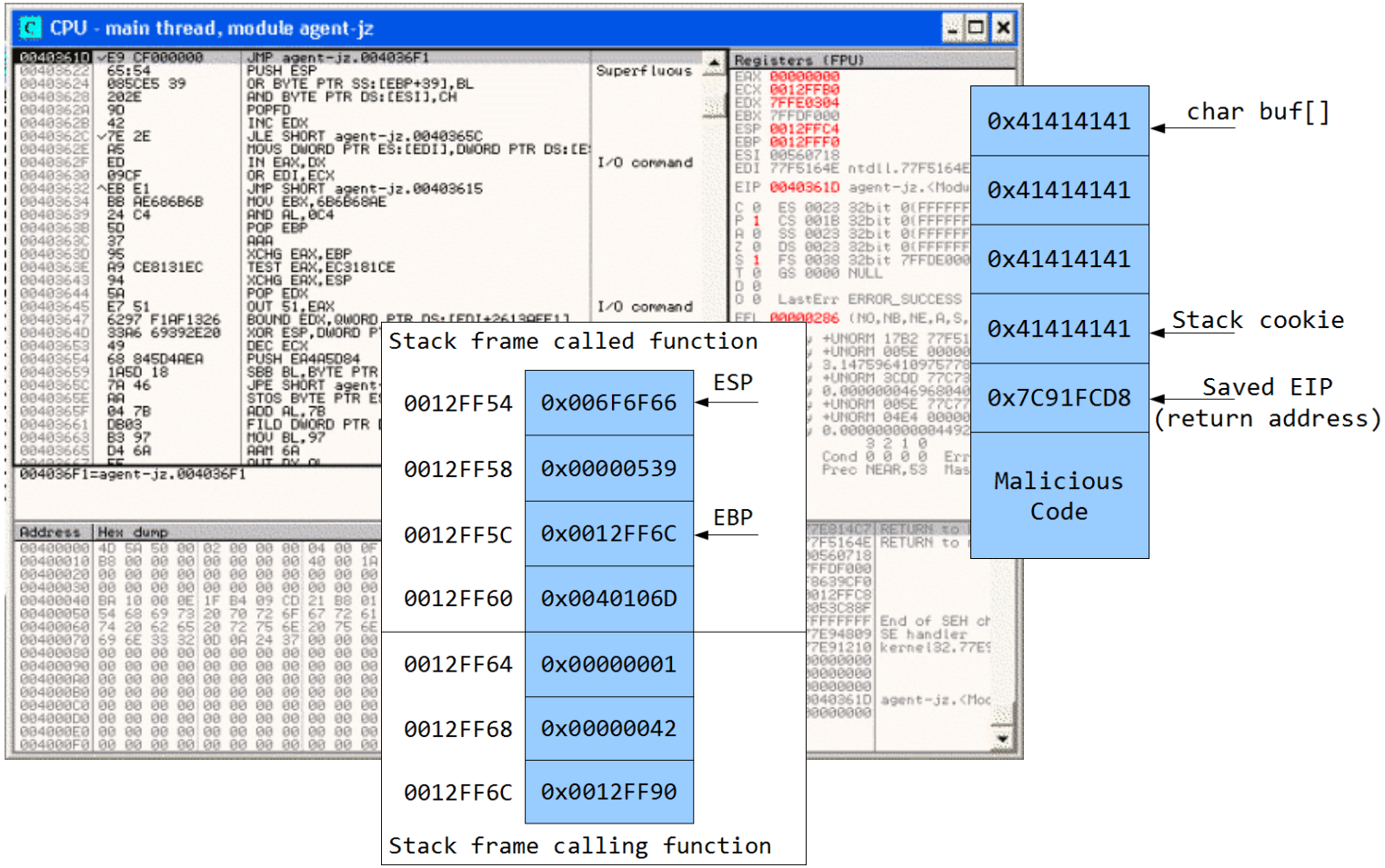
Open Letter to RSA Customers

Arthur W. Coviello, Jr.

Like any large company, EMC experiences and successfully repels multiple cyber attacks on its IT infrastructure every day. Recently, our security systems identified an extremely sophisticated cyber attack in progress being mounted against RSA. We took a variety of aggressive measures against the threat to protect our business and our customers, including further hardening of our IT infrastructure. We also immediately began an extensive investigation of the attack and are working closely with the appropriate authorities.

Our investigation has led us to believe that the attack is in the category of an Advanced Persistent Threat (APT). Our investigation also revealed that the attack resulted in certain information being extracted from RSA's systems. Some of that information is specifically related to RSA's

How technical will this presentation be?



How do exploits work?

That‘s what you‘ll (hopefully) know in about 30 minutes.

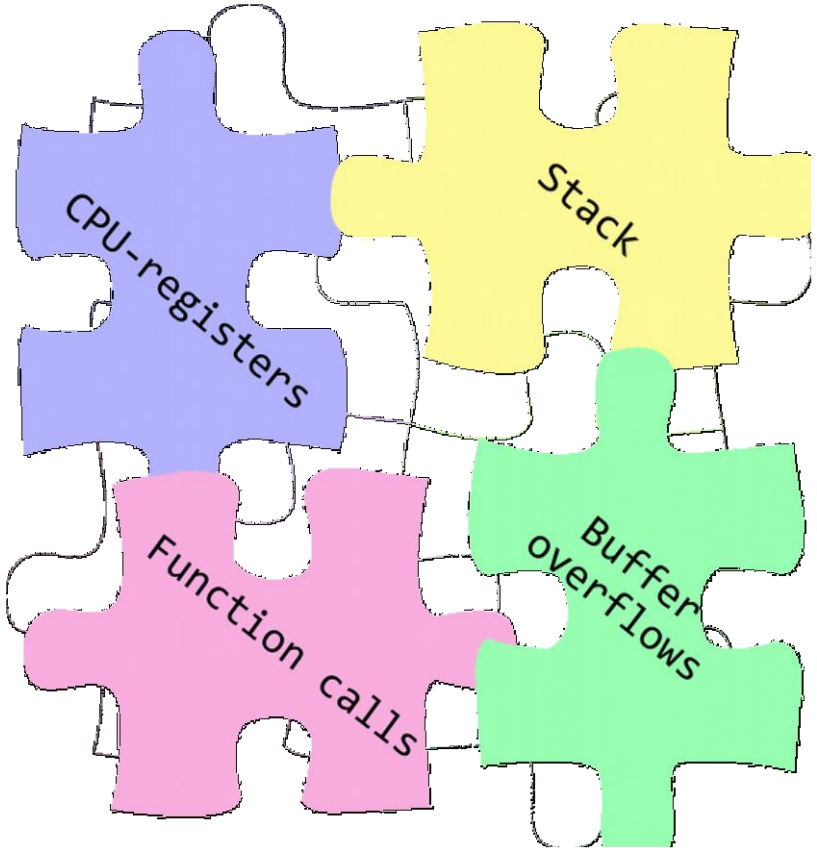
Hint: Not black magic.

Agenda

- Basics
- Simple buffer overflow exploitation
- Mitigation techniques…
- … and an example for a bypass

Basics / revision

- CPU-registers
- Stack
- Function calls
- Buffer overflows



CPU-registers

- 32 bit long (x86 architecture)
- Four general registers
  - EAX, EBX, ECX, EDX
- Four offset registers
  - ESI, EDI, ESP, EBP
- Various other registers
  - EFLAGS, EIP

- Relevant in this context
  - EIP (instruction pointer), ESP (stack pointer), EBP (base pointer / frame pointer)

Program flow / EIP

- EIP points to the next instruction to execute

- Process execution
  1. Read instruction EIP points to
  2. Increase EIP by the size of the instruction
  3. Execute previously read instruction
  4. Goto 1.

Program flow / EIP

- Example

	Address	Opcode	ASM Instruction
EIP ->	00401060	41	INC ECX
	00401061	2BC1	SUB EAX, ECX
	00401063	83C0 20	ADD EAX, 20
	00401066	33DB	XOR EBX, EBX

Program flow / EIP

- Example

```
        Address      Opcode        ASM Instruction
        00401060     41            INC ECX
EIP ->  00401061     2BC1          SUB EAX, ECX
        00401063     83C0 20       ADD EAX, 20
        00401066     33DB          XOR EBX, EBX
```

Program flow / EIP

- Example

```
        Address      Opcode        ASM Instruction
        00401060     41            INC ECX
        00401061     2BC1          SUB EAX, ECX
EIP ->  00401063     83C0 20       ADD EAX, 20
        00401066     33DB          XOR EBX, EBX
```

Program flow / EIP

- Example

```
        Address      Opcode        ASM Instruction
        00401060     41            INC ECX
        00401061     2BC1          SUB EAX, ECX
        00401063     83C0 20       ADD EAX, 20
EIP ->  00401066     33DB          XOR EBX, EBX
```

Program flow / EIP

- Function call

```
        Address      Opcode        ASM Instruction
        00401060     41            INC ECX
        00401061     2BC1          SUB EAX, ECX
        00401063     83C0 20       ADD EAX, 20
EIP ->  00401066     33DB          XOR EBX, EBX
        00401068     E8 9DFFFFFF   CALL 0040100A
        0040106D     41            INC ECX
```

Program flow / EIP

- Function call

	Address	Opcode	ASM Instruction
	00401060	41	INC ECX
	00401061	2BC1	SUB EAX, ECX
	00401063	83C0 20	ADD EAX, 20
	00401066	33DB	XOR EBX, EBX
EIP ->	00401068	E8 9DFFFFFF	CALL 0040100A
	0040106D	41	INC ECX

Program flow / EIP

- Function call

	Address	Opcode	ASM Instruction
	00401060	41	INC ECX
	00401061	2BC1	SUB EAX, ECX
	00401063	83C0 20	ADD EAX, 20
	00401066	33DB	XOR EBX, EBX
	00401068	E8 9DFFFFFF	CALL 0040100A
EIP ->	0040106D	41	INC ECX

Program flow / EIP

- Function call

Address	Opcode	ASM Instruction
00401060	41	INC ECX
00401061	2BC1	SUB EAX, ECX
00401063	83C0 20	ADD EAX, 20
00401066	33DB	XOR EBX, EBX
00401068	E8 9DFFFFFF	CALL 0040100A
0040106D	41	INC ECX

- EIP points to 0040100A (subroutine)

Program flow / EIP

- Function call

	Address	Opcode	ASM Instruction
	00401060	41	INC ECX
	00401061	2BC1	SUB EAX, ECX
	00401063	83C0 20	ADD EAX, 20
	00401066	33DB	XOR EBX, EBX
	00401068	E8 9DFFFFFF	CALL 0040100A
EIP ->	0040106D	41	INC ECX

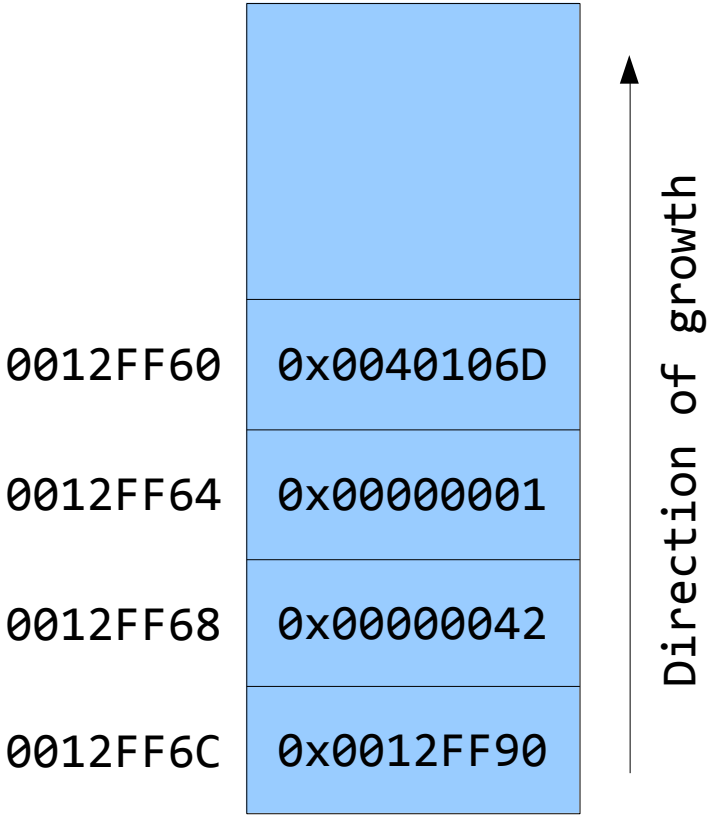
- EIP points to 0040100A (subroutine)
- When sub has finished, EIP should point to 0040106D

Program flow / EIP

- Program flow is not linear
- Content of registers has to be stored before function is called
- Content of registers has to be restored when function finishes

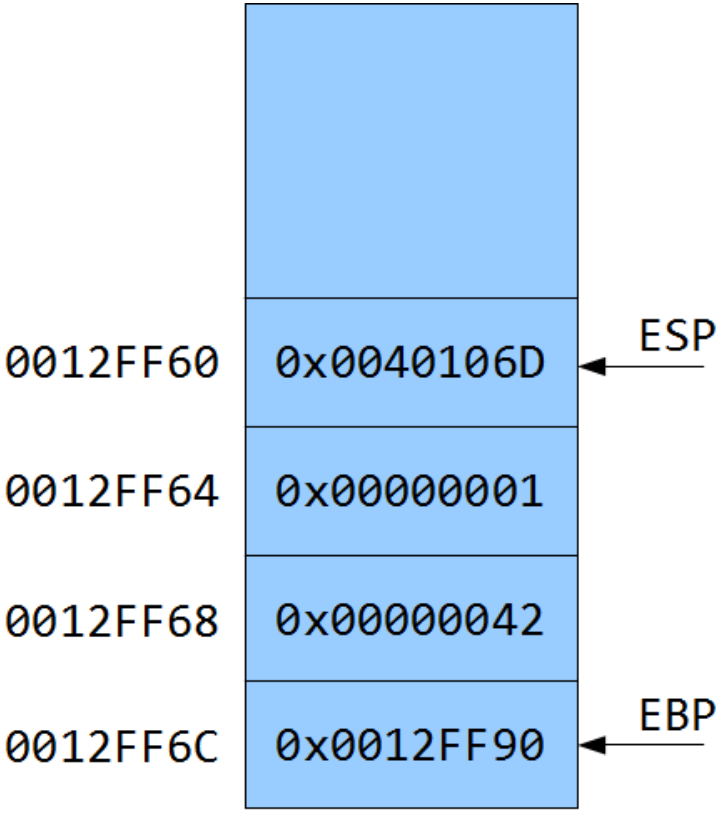
Stack

- Data structure in memory
- Used to store data temporarily (e.g. local variables, values of registers)
- Variable in size, i.e. grows and shrinks at runtime
- Grows towards lower addresses



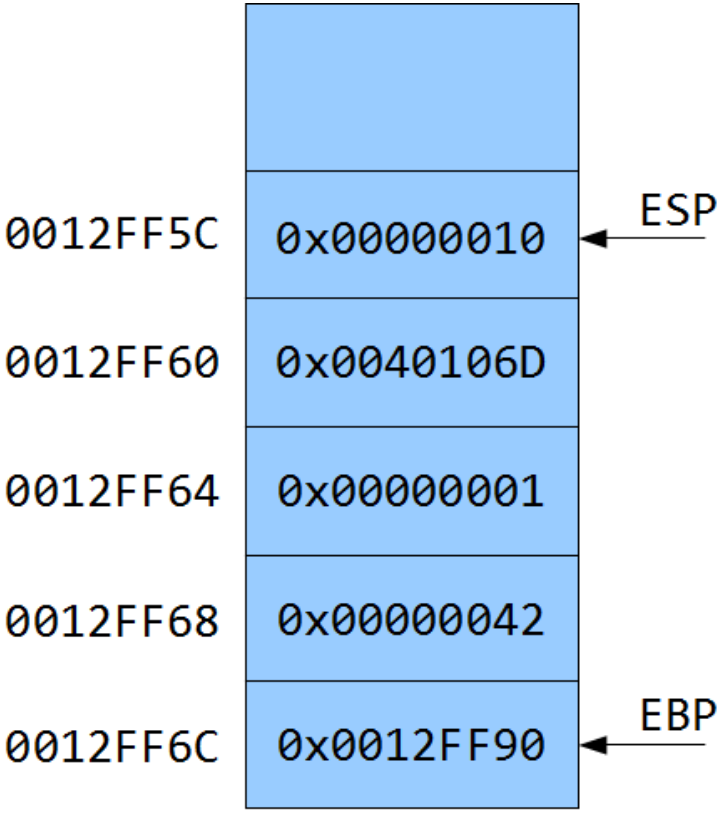
Stack

- Access via PUSH / POP instruction
- EBP = Bottom of the stack
- ESP = Top of the stack



Stack

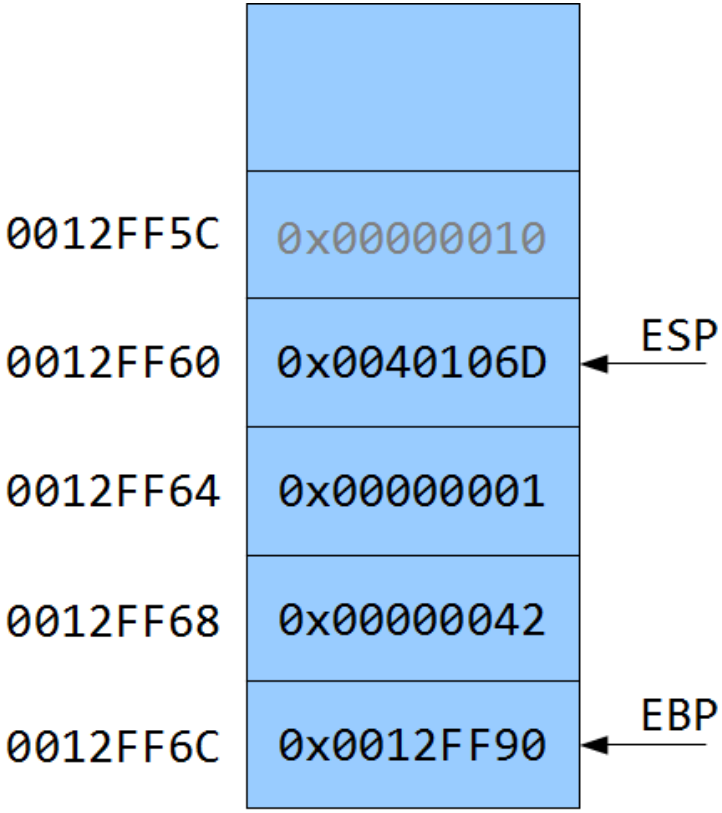
```
PUSH 10
```



Stack

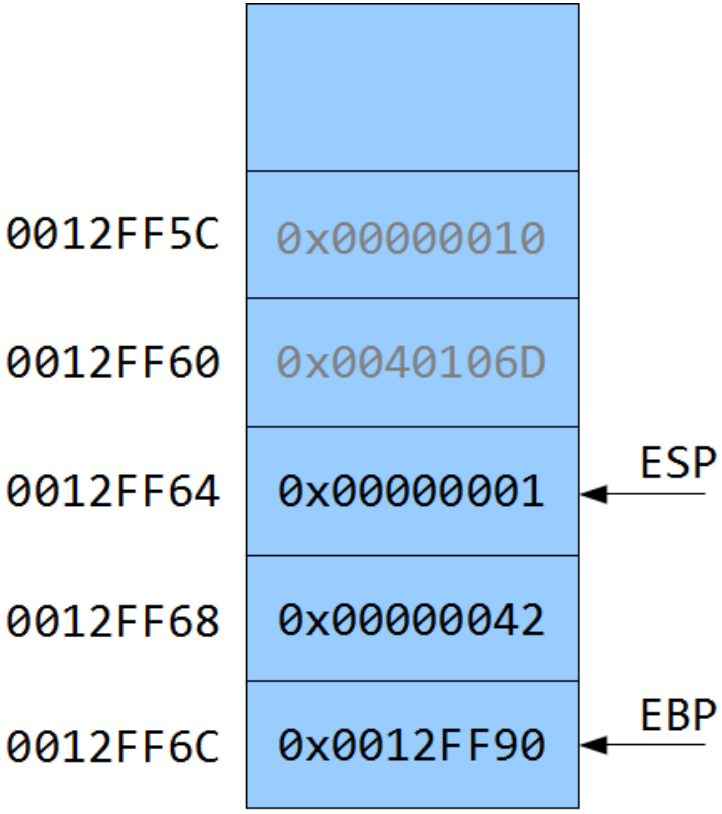
```
PUSH 10

POP EAX
```



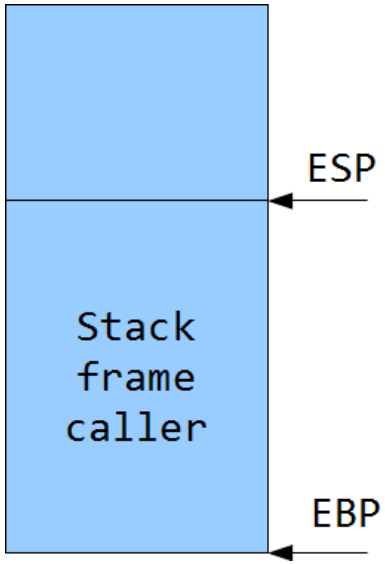
Stack

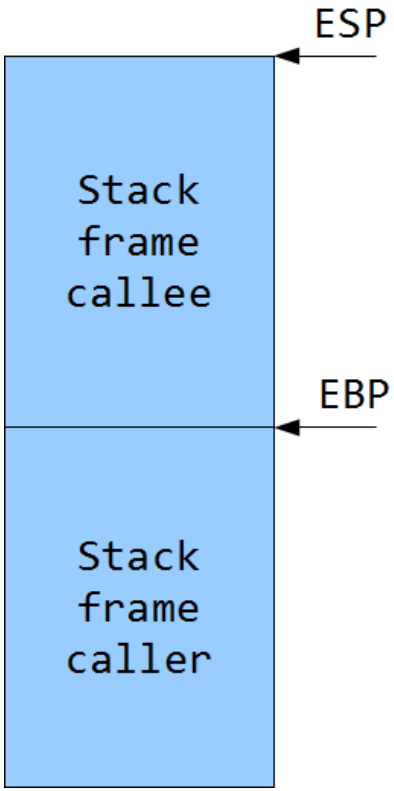
```
PUSH 10
POP EAX
POP EDX
```



Stack frames

- Space between EBP and ESP is called stack frame
- Used by the current function
- Every function has its own stack frame





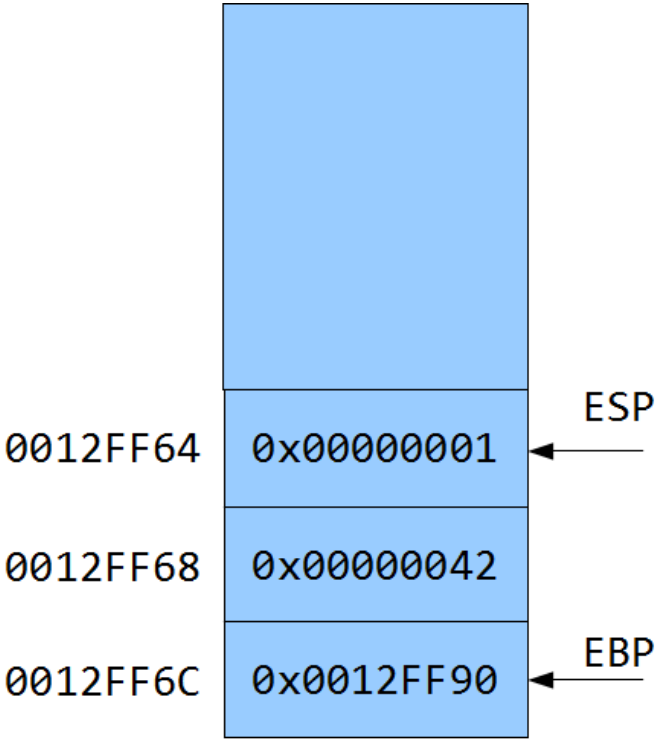
Function calls

- Store / restore registers
  - EIP (return into the calling function)
  - EBP (set the stack frame)
  - ESP does not need to be stored

- Function prologue / epilogue take care of that

Function calls

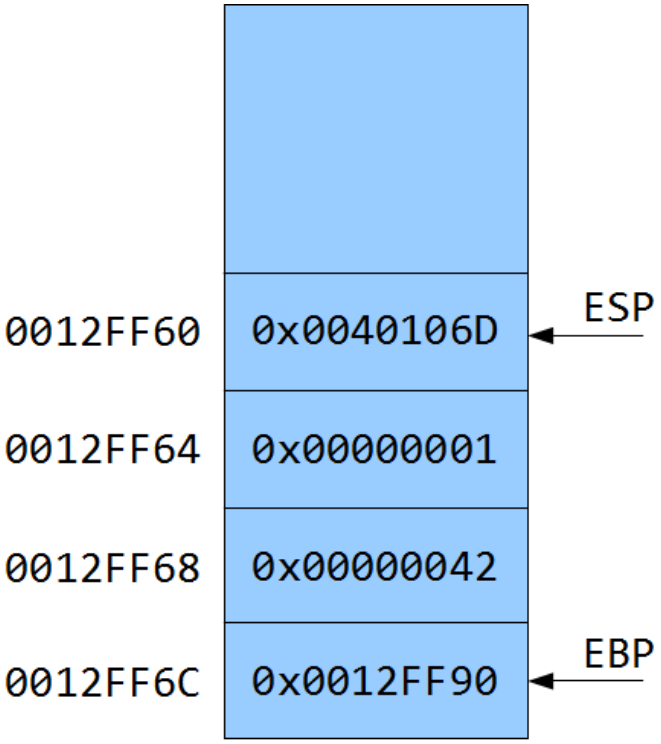
- Function prologue



Function calls

- Function prologue
  - `PUSH EIP`

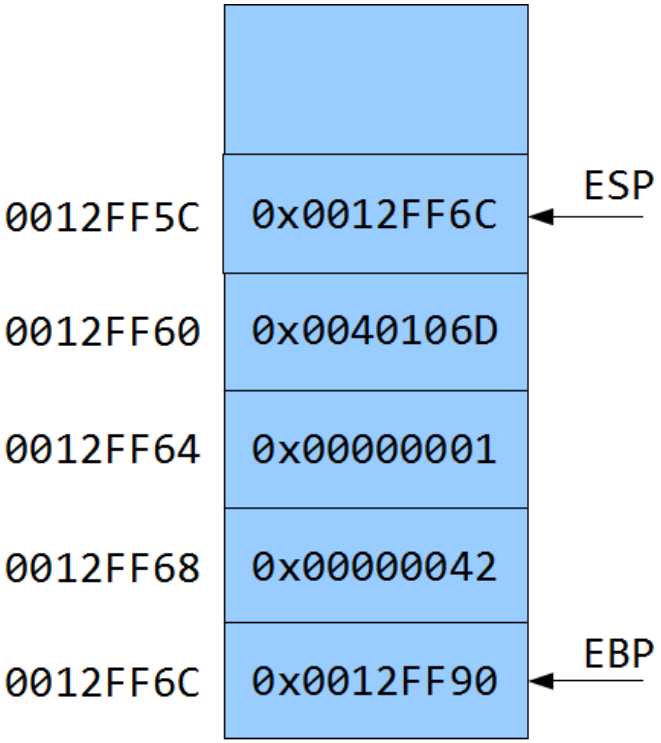
    („return address“)



Function calls

- Function prologue
  - `PUSH EIP`

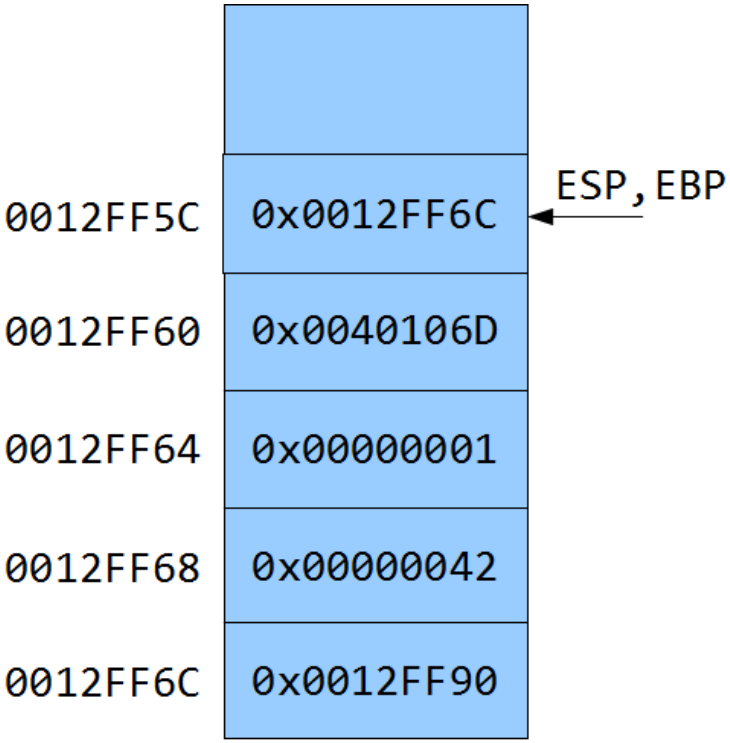
    („return address“)
  - `PUSH EBP`



Function calls

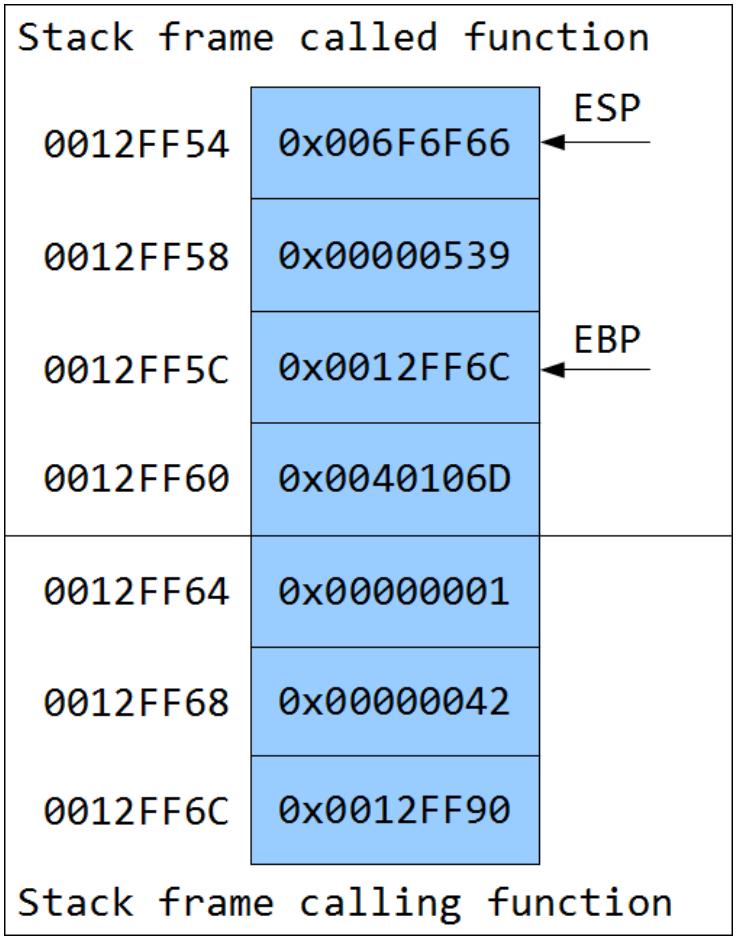
- Function prologue
  - `PUSH EIP`

    („return address“)
  - `PUSH EBP`
  - `MOV EBP, ESP`



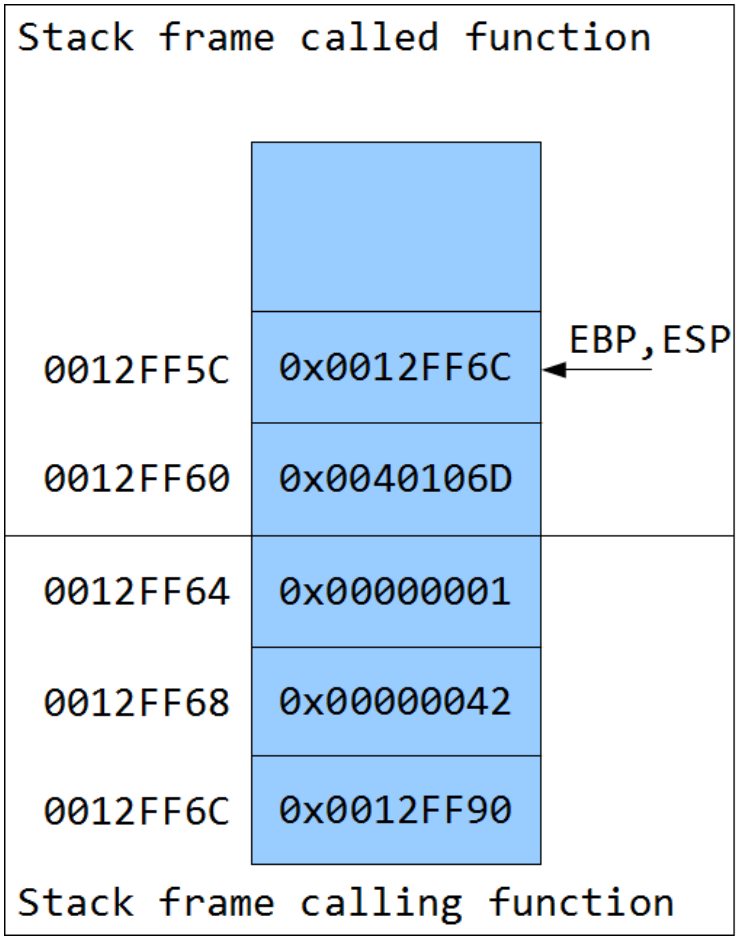
Function calls

- Function epilogue
  - `MOV ESP, EBP`



Function calls

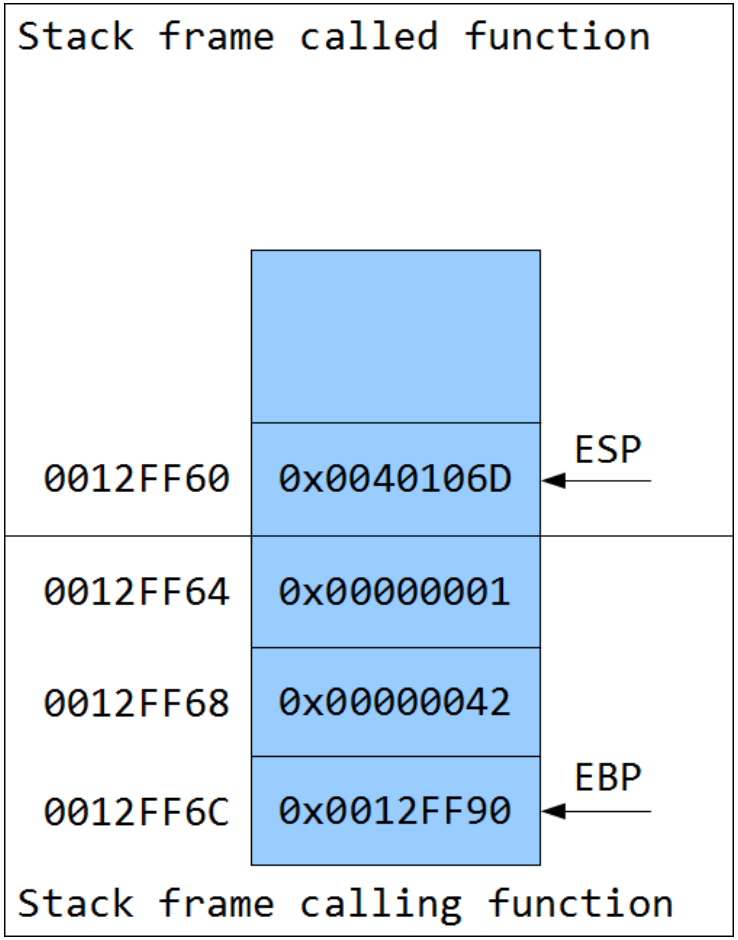
- Function epilogue
  - `MOV ESP, EBP`
  - `POP EBP`



Function calls

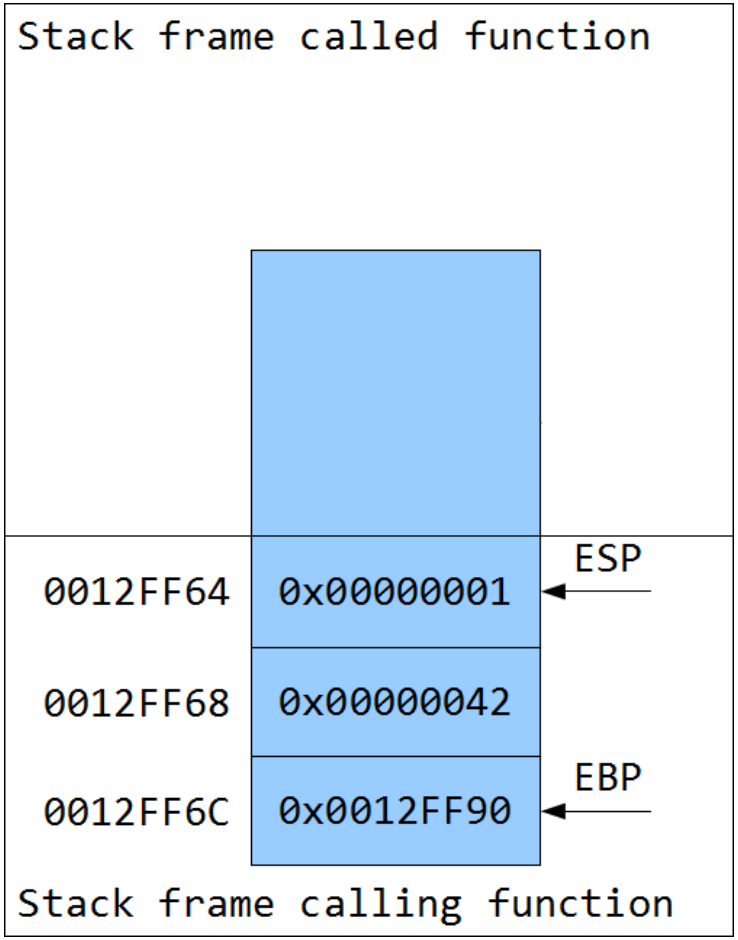
- Function epilogue
  - `MOV ESP, EBP`
  - `POP EBP`
  - `RET`

    `(= POP EIP)`



Function calls

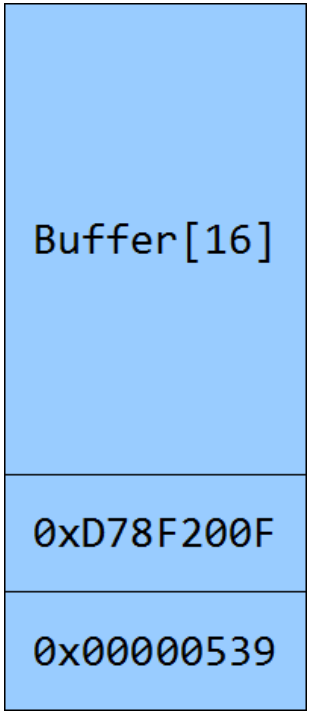
- EIP, ESP, EBP now contain the same addresses as before the function call



Buffer overflows

- Writing more data in an array than there‘s memory reserved

- Writing 19 ‚A‘s in Buffer…



Buffer overflows

- Writing more data in an array than there‘s memory reserved

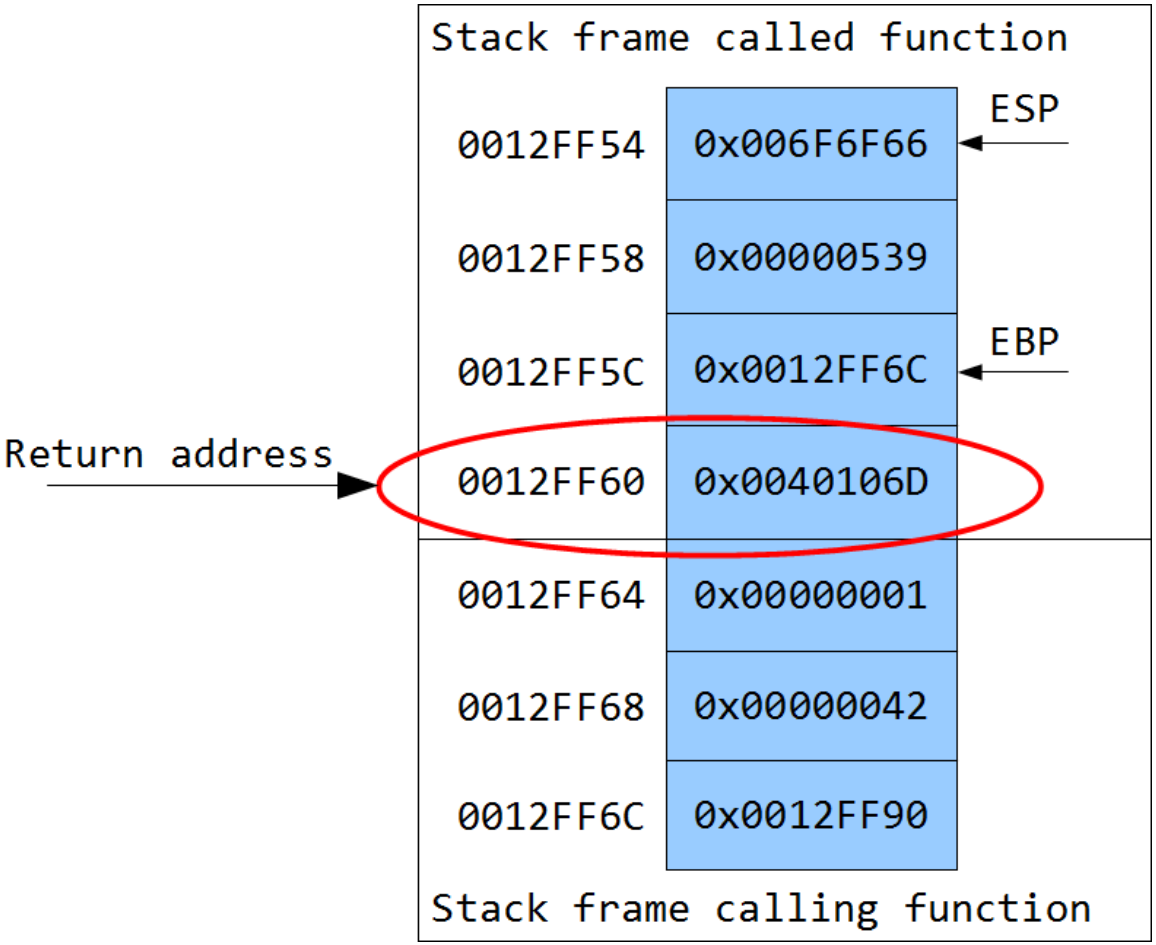
- Writing 19 ‚A‘s in Buffer…

  overwrites data in subsequent memory!

- No automatic bounds checks in C / C++
  → programmer‘s responsibility

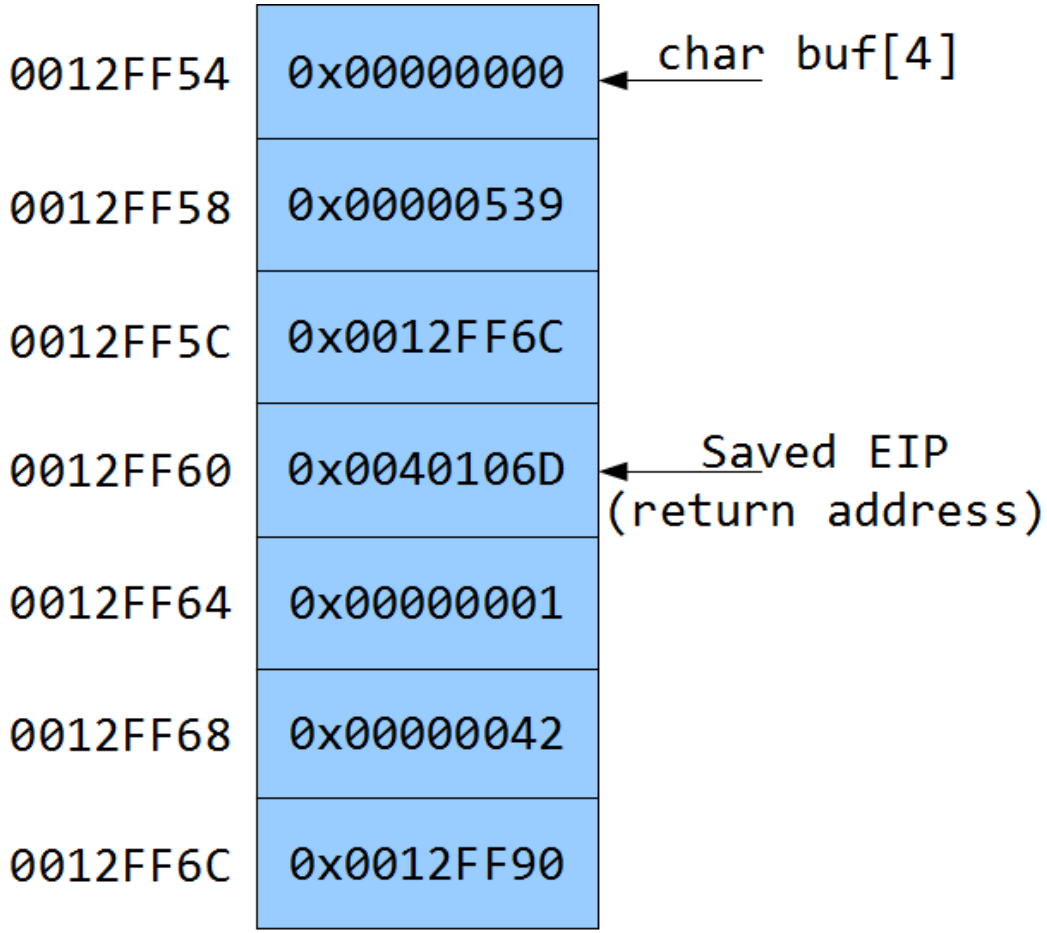
0x41414141
0x41414141
0x41414141
0x41414141
0x00414141
0x00000539

Basics / revision – the essential

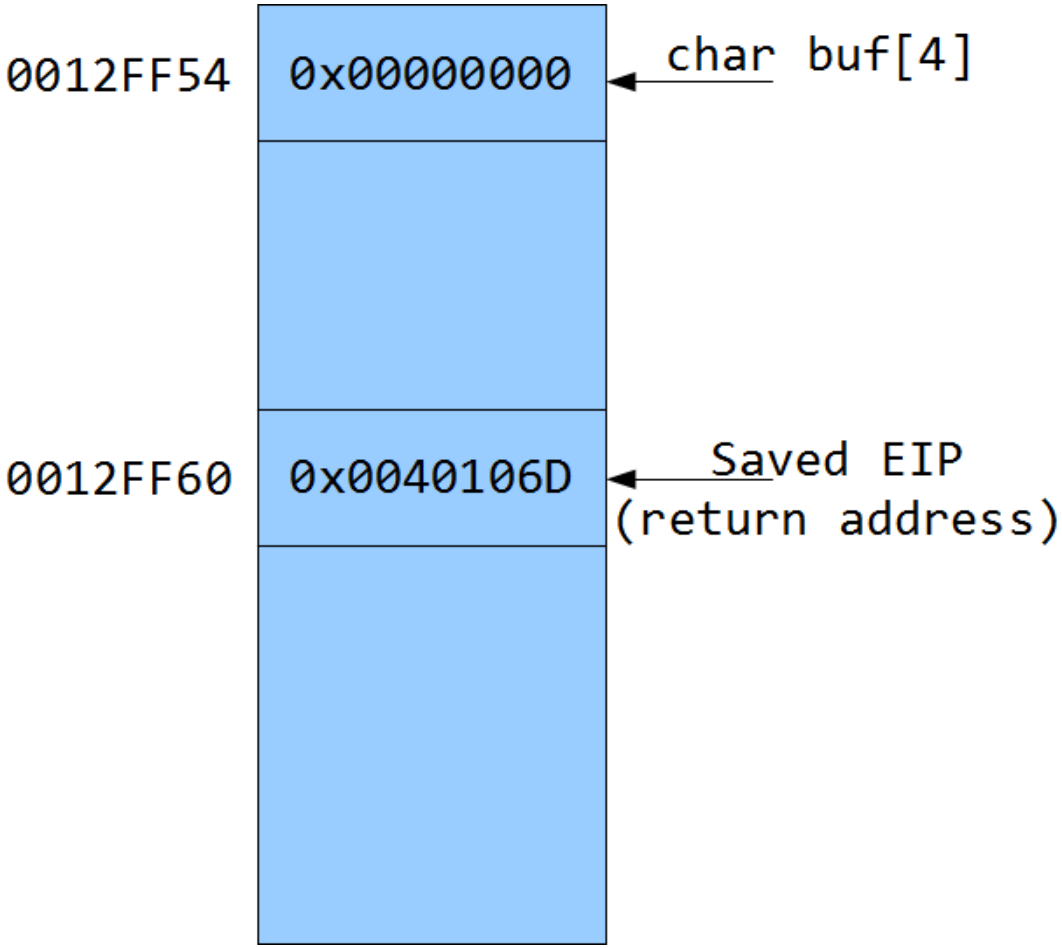


QUESTIONS SO FAR?

Basic buffer overflow exploitation



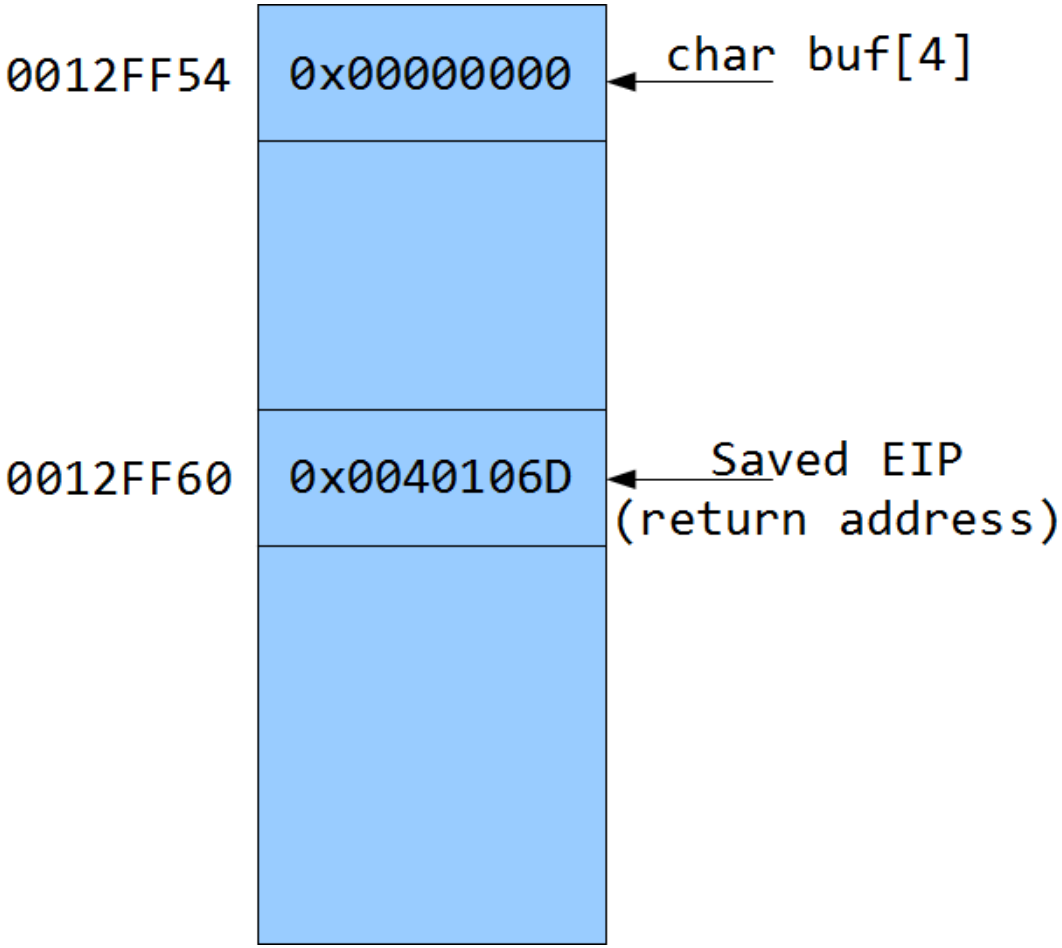
Basic buffer overflow exploitation



Vulnerable function contains

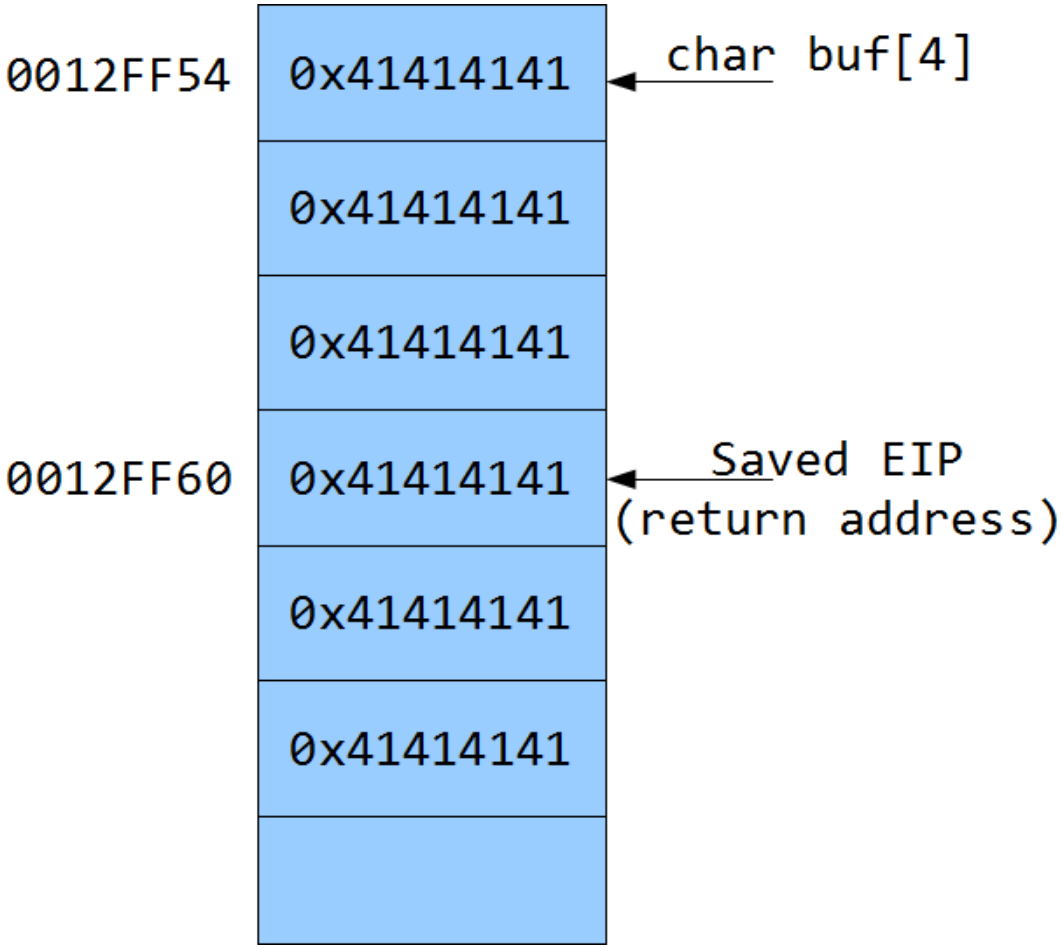
```
[…]
strcpy(buf, in);
return;
[…]
```

Basic buffer overflow exploitation



Filling the buffer with 24 ‚A‘s…

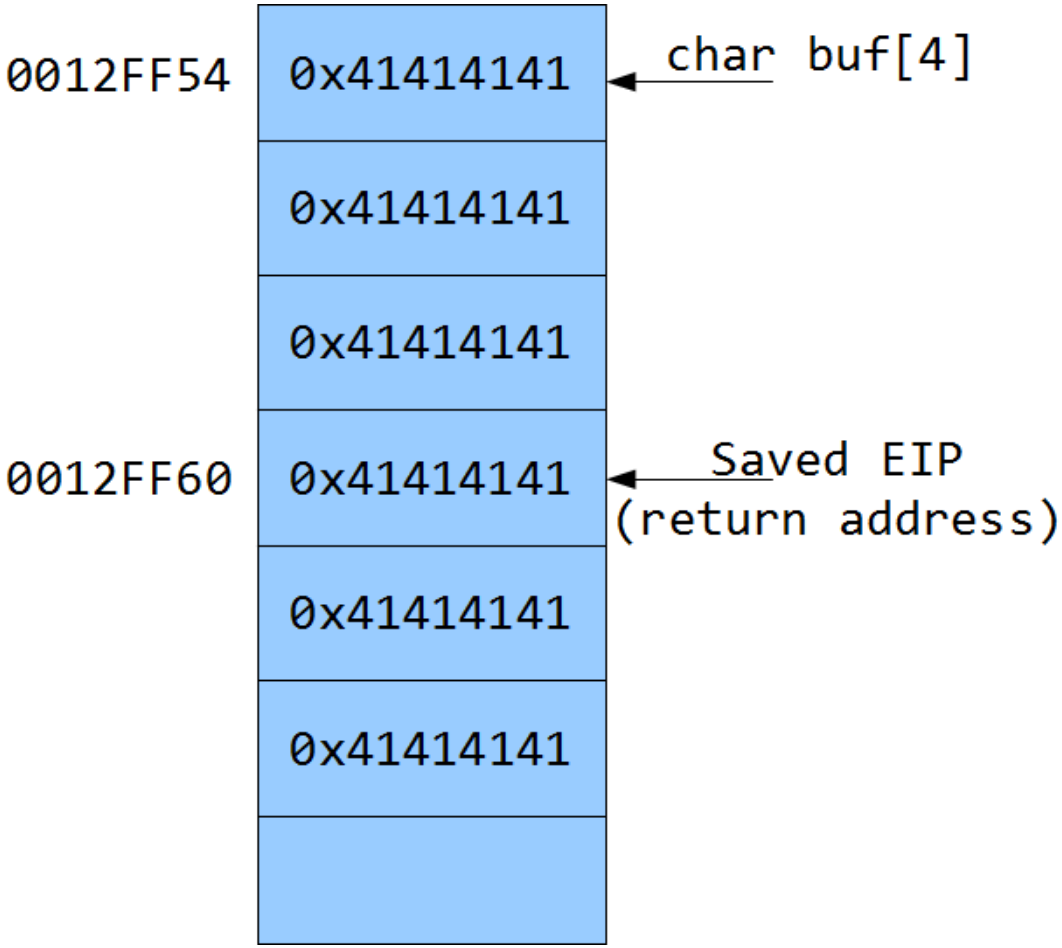
Basic buffer overflow exploitation



Filling the buffer with 24 ‚A‘s

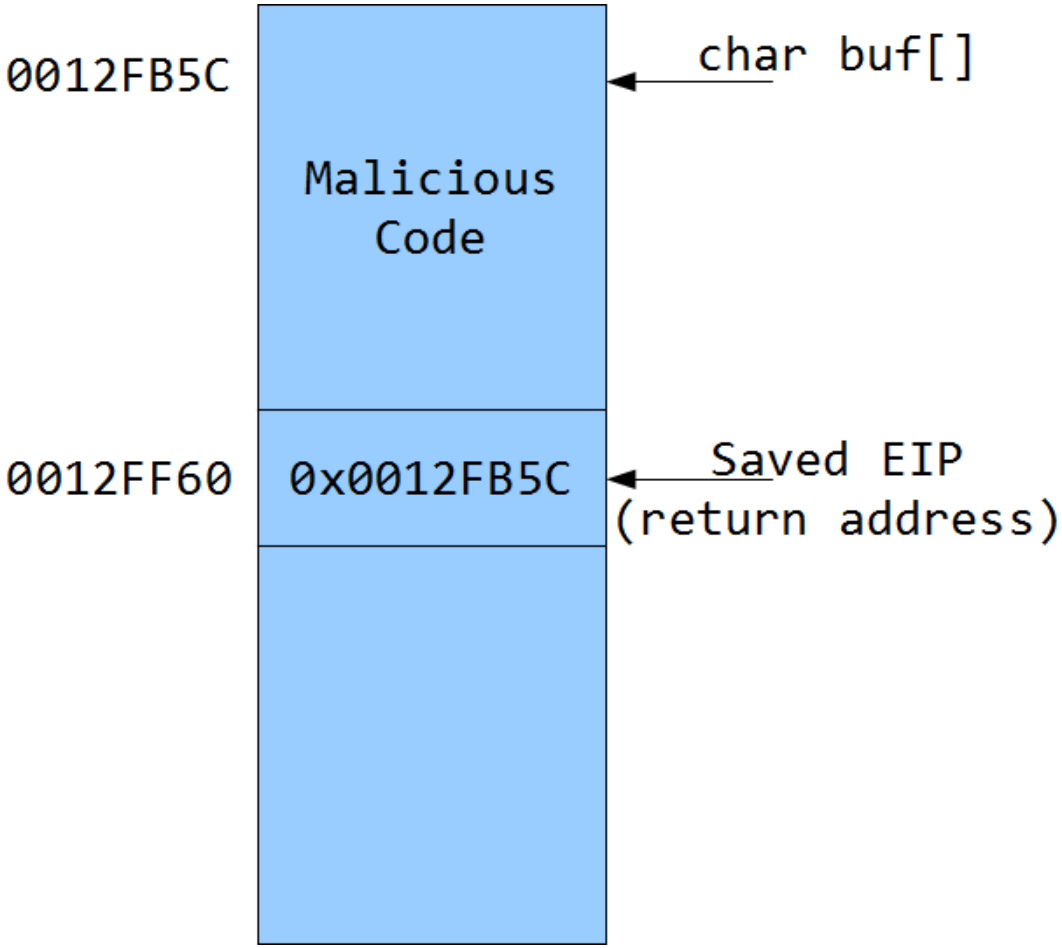
… overwrites the return address on the stack!

Basic buffer overflow exploitation



RET will load `0x41414141` into EIP, the CPU will try to execute whatever is stored at this location → program flow can be diverted

Basic buffer overflow exploitation



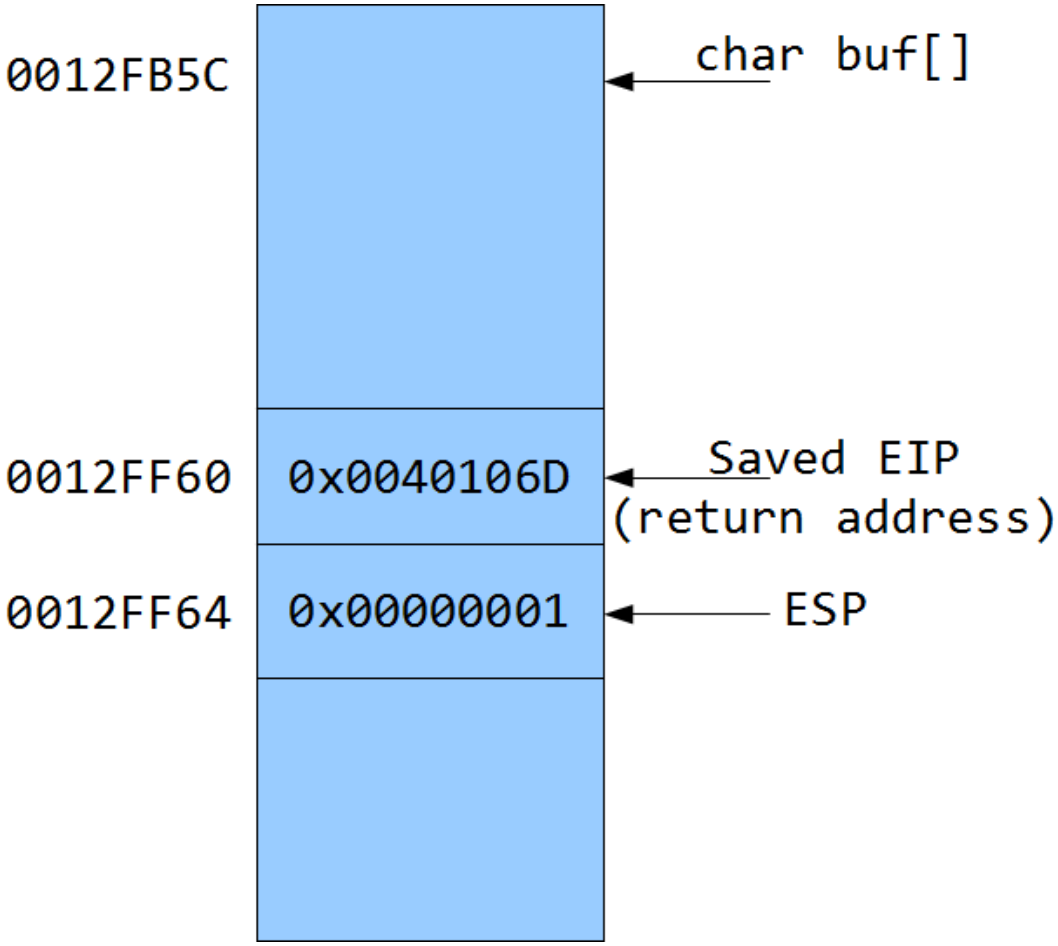
RET will load `0x0012FB5C` into EIP, the CPU will execute the malicious code!

Basic buffer overflow exploitation

- Basic principle:
  - Put code on stack
  - Exploit buffer overflow vulnerability to overwrite saved EIP to jump to the code

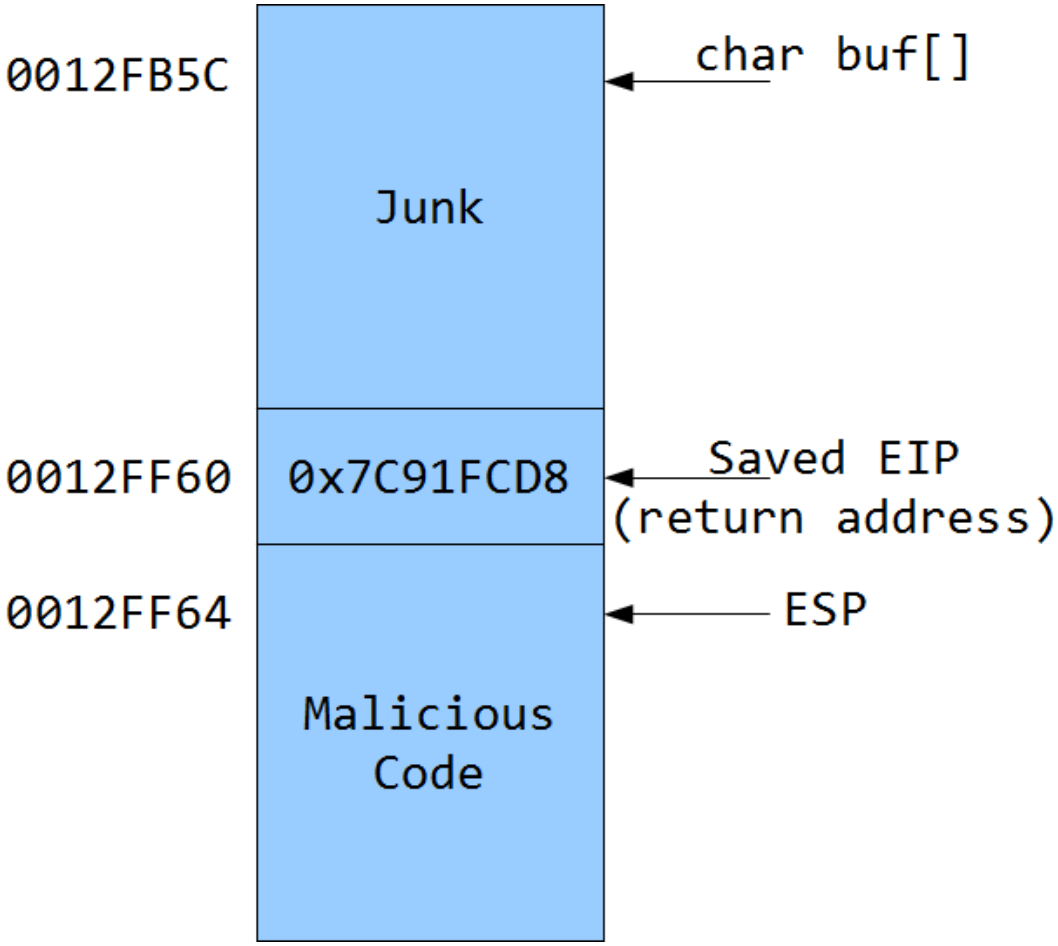
- Various ways to jump to the code
  - Hard-coding jump address
  - Using a register which points to the code

Basic buffer overflow exploitation



Stack after RET – ESP points to the address after the return address

Basic buffer overflow exploitation



- Fill ‘buf’ with junk
- Overwrite return address with address of `JMP ESP` instruction
- Put malicious code after return address (where ESP points to)

Basic buffer overflow exploitation

- Structure of this exploit
  - 1028 x ‘A’ (fills the buffer)
  - Address of `JMP ESP` instruction (overwrites the return address)
  - Malicious code

- More elegant and robust solution which is immune to slight variations of stack addresses across systems

QUESTIONS AT THIS POINT?

Bytecode

- Other names: shellcode, payload
- Code based on opcodes of assembly instructions

- So it’s basically ASM, hence can perform any given task a program can

Bytecode

- Example:

```
\x8b\xec\x68\x65\x78\x65\x20\x68\x63\x6d\x64\x2e
\x8d\x45\xf8\x50\xb8\x8D\x15\x86\x7C\xff\xd0
```

Opcode	ASM Instruction
8BEC	MOV EBP,ESP
68 65786520	PUSH 20657865
68 636D642E	PUSH 2E646D63
8D45 F8	LEA EAX,DWORD PTR SS:[EBP-8]
50	PUSH EAX
B8 8D15867C	MOV EAX,kernel32.WinExec
FFD0	CALL EAX

Bytecode

- Must not contain certain characters like \x00 (string terminator)
  - Solution: encoders
- Needs to be as small as possible
  - Solution: egg-hunters / omelett hunters

Mitigation techniques

- Goal: Make reliable exploitation more difficult and time consuming

- Non-executable stack (DEP / NX bit / XD bit / XN / Enhanced Virus Protection, …)
- Address space layout randomization (ASLR)
- Stack cookie / canary

Non-executable stack

- Around since ~ 2004
- OS and compiler dependant

- Marks the stack as non-executable → trying to execute bytecode from the stack crashes the program

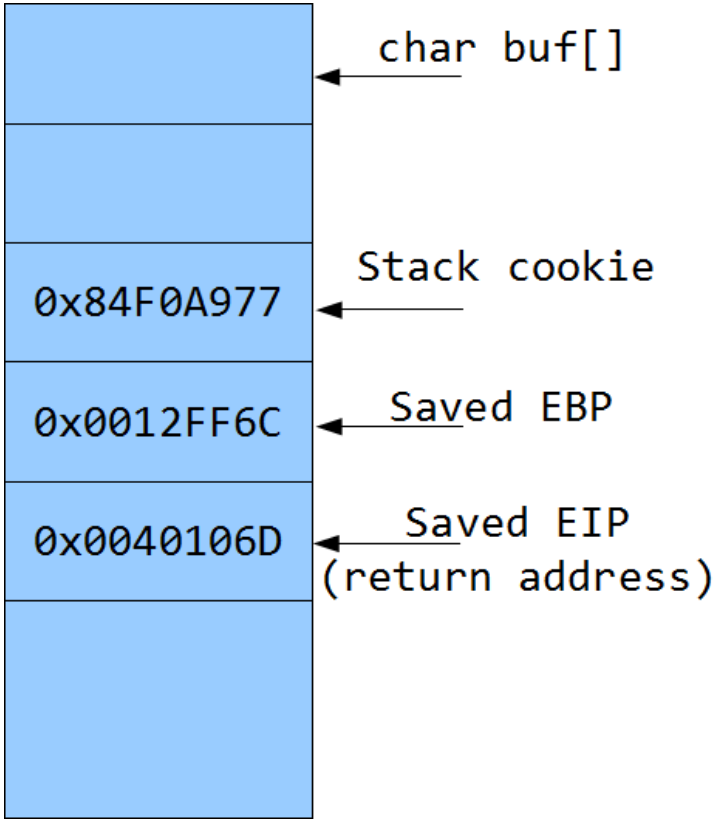
Address space layout randomization

- Around since ~ 2005
- OS and compiler dependant

- Randomizes memory location of binaries, stack, libraries etc. → hard-coding addresses in the exploit doesn't work anymore

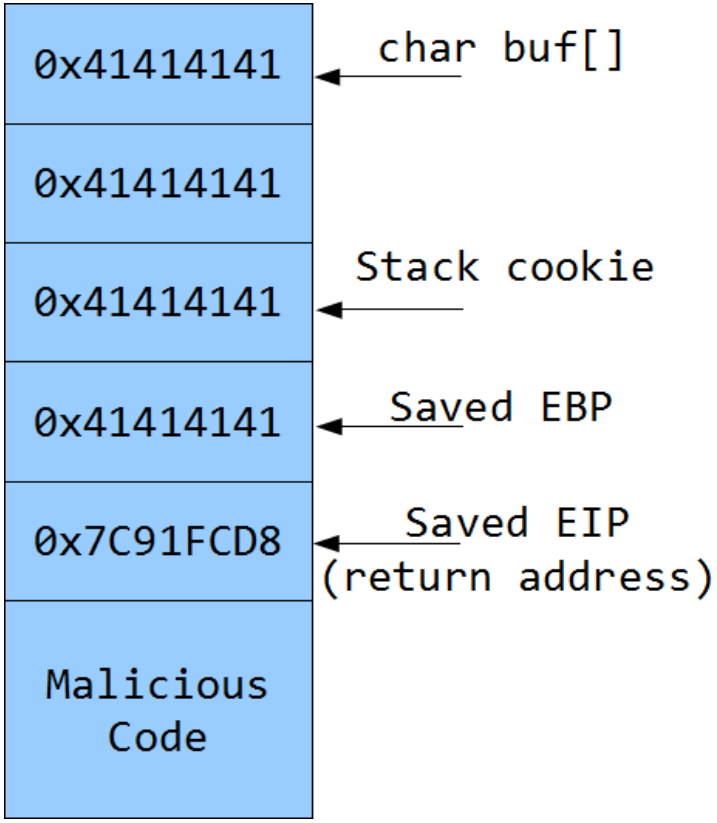
Stack cookie / canary

- Compiler dependant
- Pseudo-random four byte value on the stack before the saved EBP and the return address
- Function prologue and epilogue are instrumented to insert / check cookie



Stack cookie / canary

- Overwriting the return address also overwrites the cookie → program aborts before RET instruction is executed



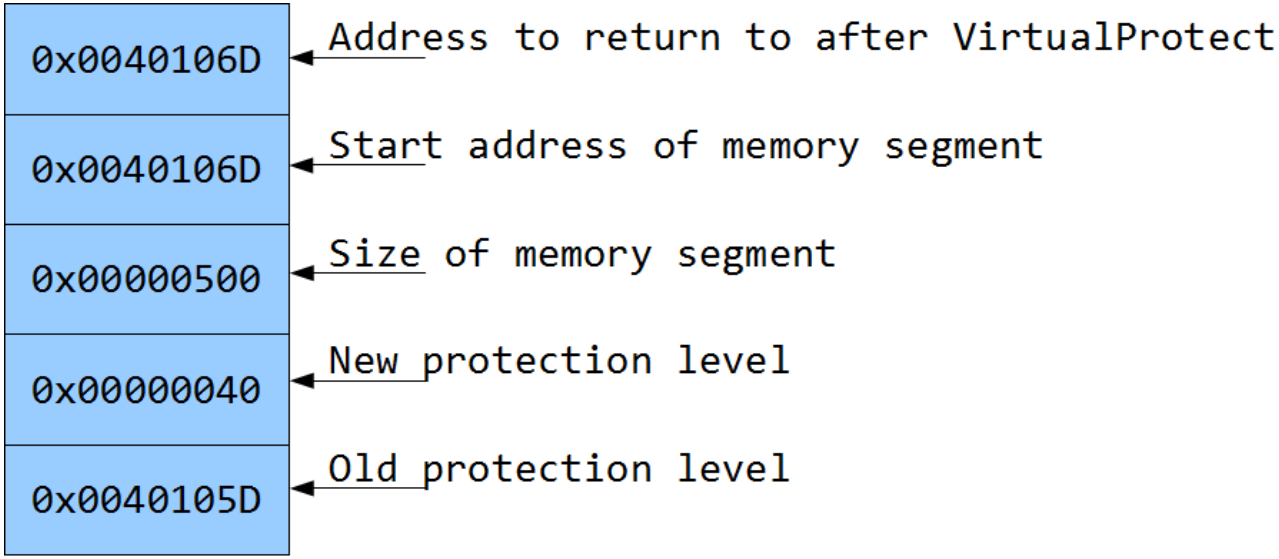
Bypassing a non-executable stack

- Problem: the stack is marked as non-executable, so the payload won‘t run

- (One) solution: change permissions of parts of the stack at runtime using OS functions

Bypassing a non-executable stack

- Use VirtualProtect() in ntdll.dll to change access level
- Ultimately, the stack needs to look like this:



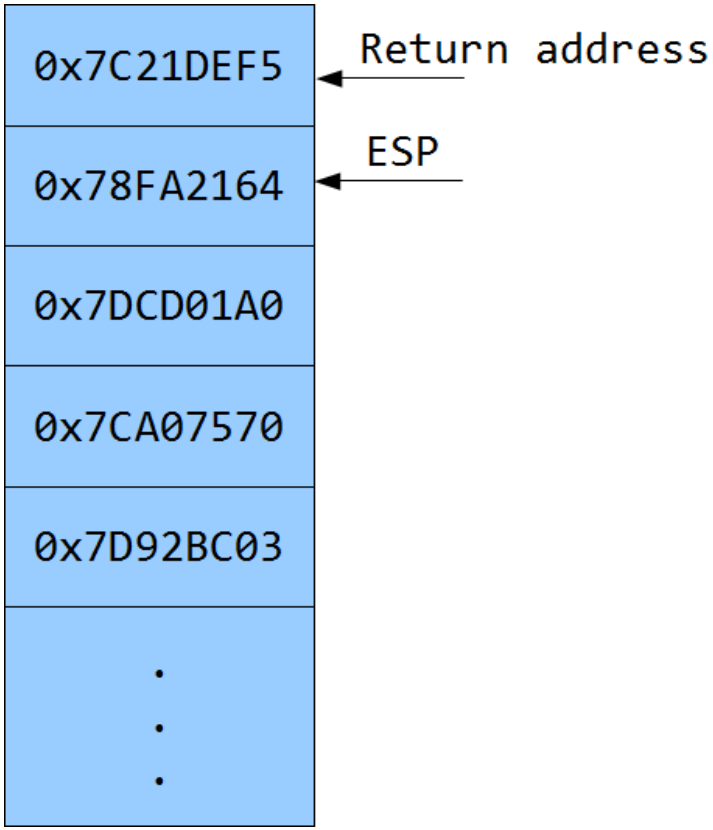
Bypassing a non-executable stack

- „Hey, those parameters contain nullbytes!“

  → Problem: parameters can‘t be hard-coded in the exploit
- Solution: create them dynamically
  - Instead of hard-coding `0x00000040` in the exploit, use

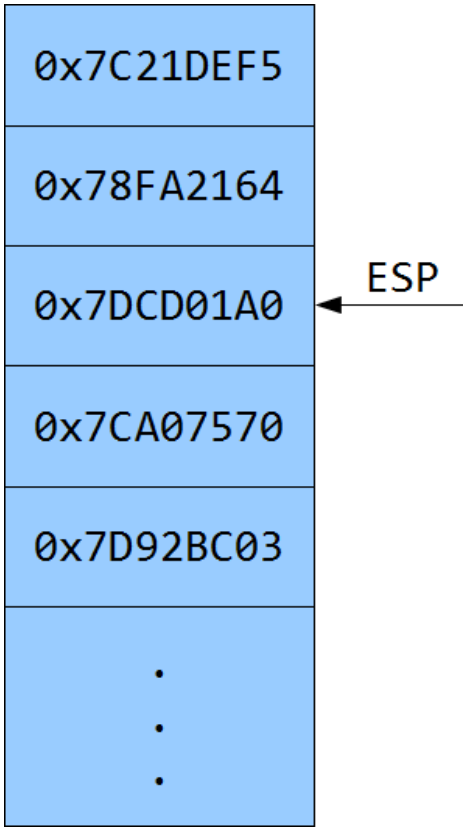
    `XOR EAX, EAX, ADD EAX 40, PUSH EAX`
- Problem: non-executable stack
- Solution: return-oriented programming (ROP)
  - Utilize short sequences of instructions („gadgets“) which end with RET from other loaded libraries

Return-oriented programming




```
7C21DEF5: […]
7C21DEF7: […]
7C21DEF8: RET
```

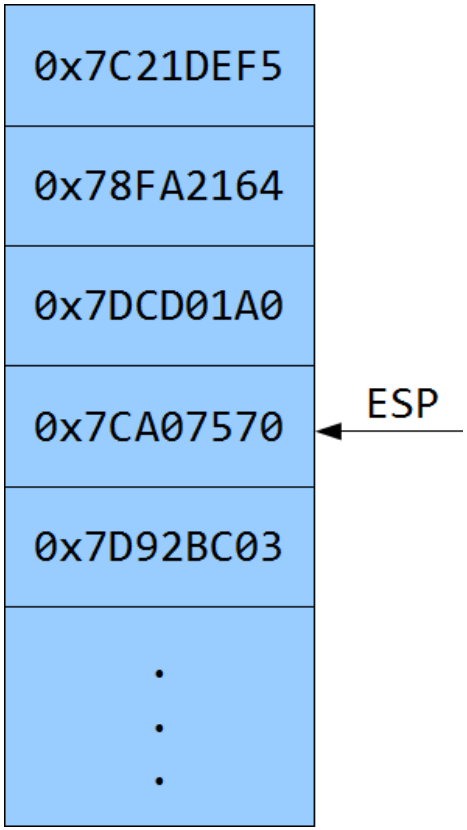
Return-oriented programming




```
7C21DEF5: […]
7C21DEF7: […]
7C21DEF8: RET

78FA2164: […]
78FA2169: RET
```

Return-oriented programming

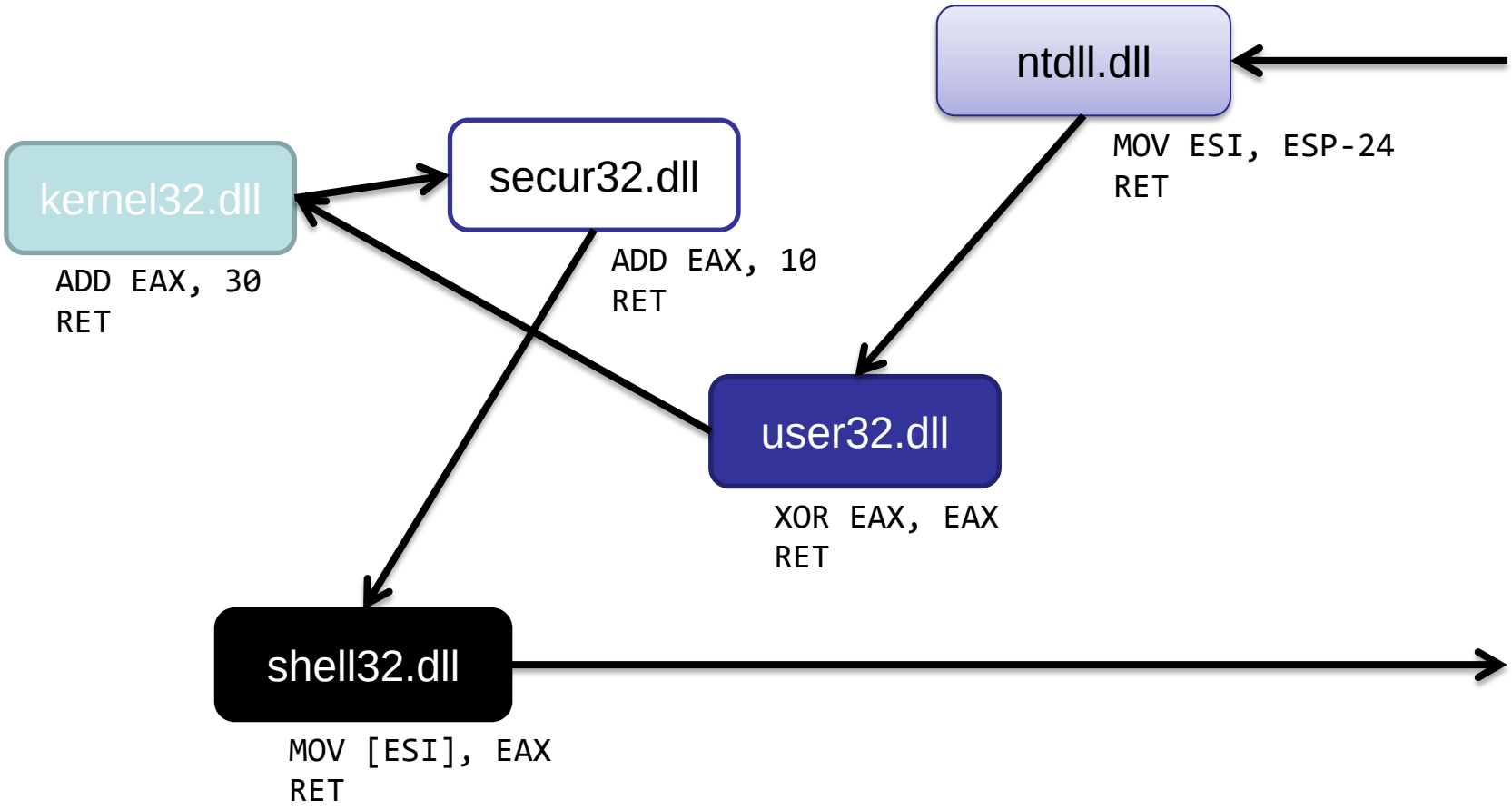



```
7C21DEF5: […]
7C21DEF7: […]
7C21DEF8: RET

78FA2164: […]
78FA2169: RET

7DCD01A0: […]
7DCD01A3: RET
```

Return-oriented programming



Return-oriented programming

Challenges:

„Clean“ gadgets are very rare!

```
MOV [ESI], EAX
RET
```

Would be nice, but unlikely…

```
MOV [ESI+24], EAX
MOV EAX, EBX
POP ESI
RET
```

More realistic – most gadgets will affect various other registers.

Conclusion

- Buffer overflow vulnerabilites are still a big threat
- Raise programmers‘ awareness
- Come up with new solutions that actually work

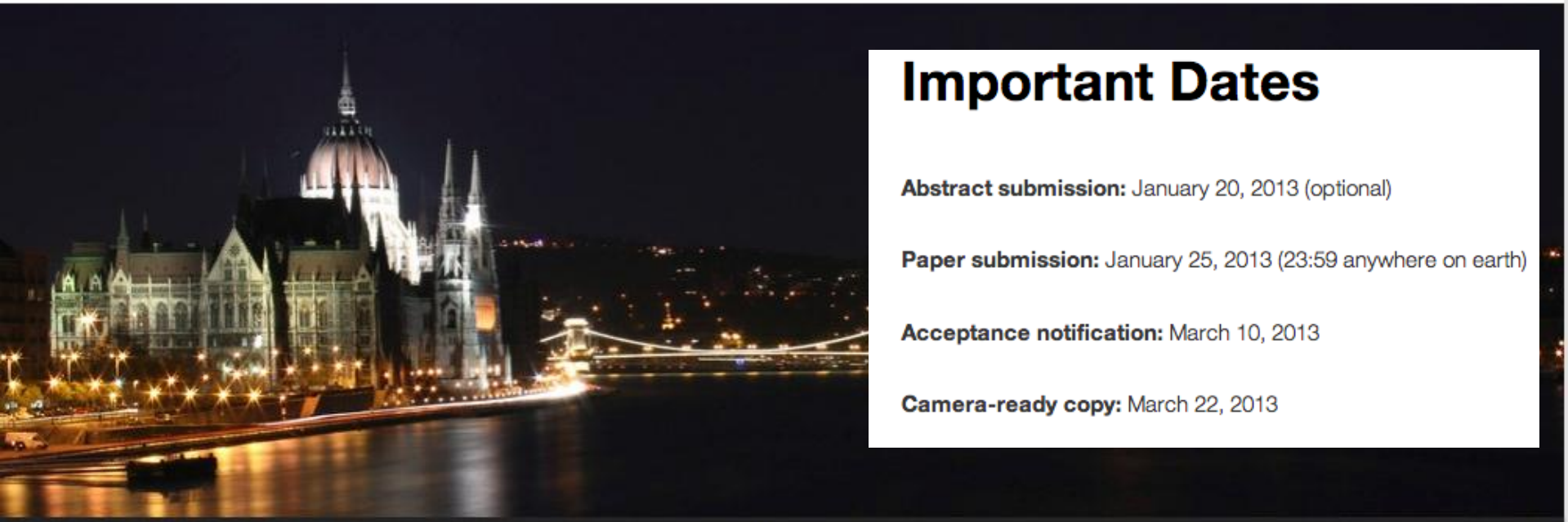
Further reading

- Aleph One – Smashing the Stack for Fun and Profit (http://www.phrack.com/issues.html?issue=49&id=14)
- Jon Erickson – Forbidden Code / Hacking: The Art of Exploitation
- http://uninformed.org/
- http://corelan.be/
- http://www.exploit-db.com/
- http://www.thegreycorner.com/

International Conference on Software Composition 2013

Budapest, Hungary, between June 17th and 21st 2013

Search



Important Dates

Abstract submission: January 20, 2013 (optional)

Paper submission: January 25, 2013 (23:59 anywhere on earth)

Acceptance notification: March 10, 2013

Camera-ready copy: March 22, 2013

Home | Call for Papers | Submission | **Important Dates** | Venue | Organization | Previous SCs

http://sc2013.ec-spride.de/

EC SPRIDE
EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN