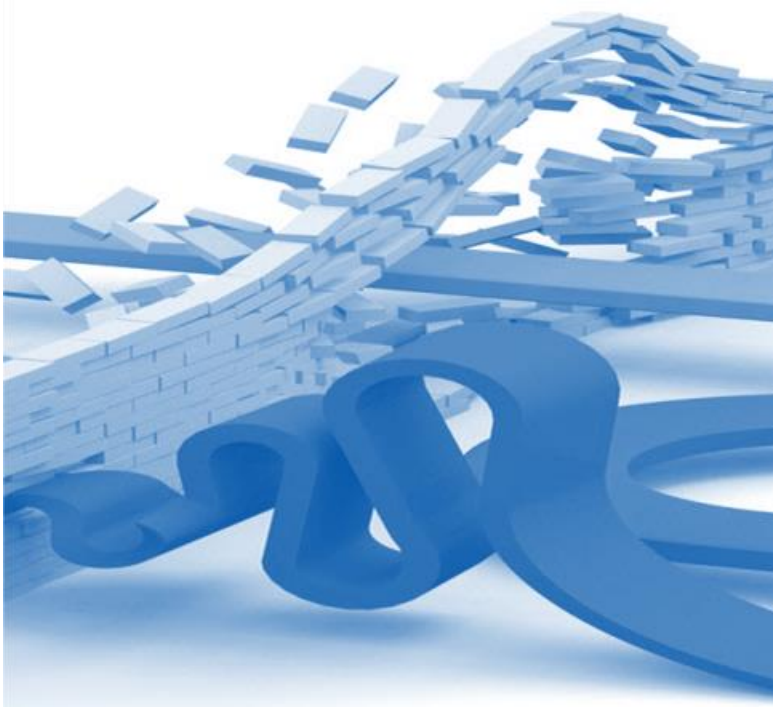


EXPLOITATION

Prof. Dr. Bernhard Tellenbach

Content

- Definition and classification of exploits
- Revisited: Stack-based buffer overflow attacks (=> SWS1)
- Revisited: ASLR, Stack Canaries, NX/DEP (=> SWS1)
- Return-Oriented Programming (ROP)
  - Overview
  - Details
  - Mitigations

Goals

- You know the concept of Return-Oriented Programming
- You can craft a ROP chain that achieves a specific goal
- You can explain why ROP can be used to circumvent NX/DEP
- You can explain under which circumstances it is possible to circumvent NX/DEP, ASLR and Stack Canaries

Exploits – Definition (2)

- An exploit is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a vulnerability

Examples

- Chunk of data: Ping of death
  - Oversized (and fragmented) IP packet crashing the server
  - Discovered in 1997 affecting almost all OS
  - Reappeared in 2013 in all Windows OS with IPv6 support
- Piece of software: JavaScript code exploiting a vulnerability in the browser
- Sequence of commands: Exploit for CVE-2017-5521 (admin PW, Netgear router)
  - Cancel authentication when asked for credentials
    => redirect to page showing recovery token
  - Supply token to: http://.../passwordrecovered.cgi?id=TOKEN => admin pw

Ping of Death example:

https://en.wikipedia.org/wiki/Ping_of_death

The maximum packet length of an IPv4 packet including the IP header is 65,535 (2^{16} − 1) bytes, a limitation presented by the use of a 16-bit wide IP header field that describes the total packet length.

The underlying Data Link Layer almost always poses limits to the maximum frame size (See MTU). In Ethernet, this is typically 1500 bytes. In such a case, a large IP packet is split across multiple IP packets (also known as IP fragments), so that each IP fragment will match the imposed limit. The receiver of the IP fragments will reassemble them into the complete IP packet, and will continue processing it as usual. When fragmentation is performed, each IP fragment needs to carry information about which part of the original IP packet it contains. This information is kept in the Fragment Offset field, in the IP header. The field is 13 bits long, and contains the offset of the data in the current IP fragment, in the original IP packet. The offset is given in units of 8 bytes. This allows a maximum offset of 65,528 ((2^{13}-1)*8). Then when adding 20 bytes of IP header, the maximum will be 65,548 bytes, which exceeds the maximum frame size. This means that an IP fragment with the maximum offset should have data no larger than 7 bytes, or else it would exceed the limit of the maximum packet length. A malicious user can send an IP fragment with the maximum offset and with much more data than 8 bytes (as large as the physical layer allows it to be).

When the receiver assembles all IP fragments, it will end up with an IP packet which is larger than 65,535 bytes. This may possibly overflow memory buffers which the receiver allocated for the packet, and can cause various problems.

Piece of Software example:

An example of a vulnerability in Flash that makes use of JIT compiler specifics and a vulnerability in a native library can be found here:
http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

Netgear router example:

https://www.trustwave.com/Resources/Security-Advisories/Advisories/TWSL2017-003/?fid=8911

When trying to access the web panel a user is asked to authenticate, if the authentication is cancelled and password recovery is not enabled, the user is redirected to a page which exposes a password recovery token. If a user supplies the correct token to the page http://router/passwordrecovered.cgi?id=TOKEN (and password recovery is not enabled), they will receive the admin password for the router.

Exploits – Definition (2)

- Exploits cause unintended or unanticipated behavior on computer software, hardware, or something electronic
- Exploits for bugs in the human hardware: Social Engineering
  - Based on specific attributes of human decision-making known as cognitive biases
- Cognitive biases are exploited in various combinations to create a variety of attack techniques like (spear) phishing or tailgating
  - Phishing: Technique of fraudulently obtaining private information
    - Phishing e-mail: message that appears to come from a legitimate business requesting "verification" of information and warning of some consequence if not provided
  - Spear phishing: Phishing but highly customized to one or a few end users
  - Tailgating: Getting entry to a restricted area by walking in behind a person who has legitimate access

Exploits – Classification

- Exploits are often classified by the action against the vulnerable system
  - Unauthorized data access, arbitrary code execution, denial of service, privilege escalation,…
- In addition, the following terms are used to characterize them:
  - A local exploit is targeting a vulnerability on the system on which it is executed, for example gain admin privileges
  - A remote exploit is targeting a vulnerability on a remote system
  - A client-side exploit targets a vulnerability of a client software
    - They often require the "help" of a user to work (open a malicious file, access a specific website etc.)
    - Drive-by attacks might use client-side exploits which do not require the help of a user (e.g., exploits triggered by malicious ads)
  - A server-side exploit targets a server and does not require the "help" of a user to work
  - 0-day/zero-day exploit: An exploit for a vulnerability that was not publicly reported, announced or is unaddressed when the exploit becomes active
    - "Day Zero": Day on which the vendor (or public) learns of the vulnerability

A **Zero-day** (also known as 0-day) vulnerability is a computer-software vulnerability that is unknown to, or unaddressed by, those who should be interested in mitigating the vulnerability (including the vendor of the target software). Until the vulnerability is mitigated, hackers can exploit it to adversely affect computer programs, data, additional computers or a network.[1] An exploit directed at a zero-day is called a zero-day exploit, or zero-day attack.

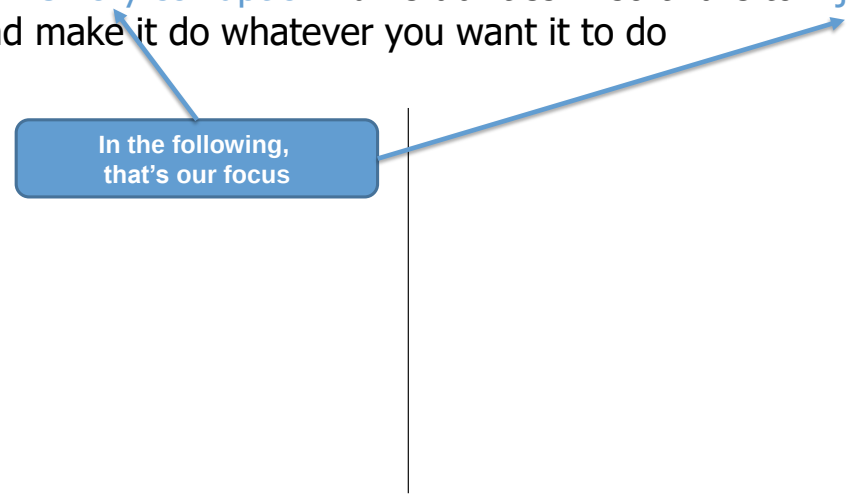
In the jargon of computer security, "Day Zero" is the day on which the interested party (presumably the vendor of the targeted system) learns of the vulnerability, leading to the vulnerability being called a "zero-day". Once the vendor learns of the vulnerability, the vendor will usually create patches or advise workarounds to mitigate it.[2]

The fewer the days since Day Zero, the higher the chance no fix or mitigation has been developed. Even after a fix is developed, the fewer the days since Day Zero, the higher is the probability that an attack against the afflicted software will be successful, because not every user of that software will have applied the fix. For zero-day exploits, unless the vulnerability is inadvertently fixed, e.g., by an unrelated update that happens to also obviate the need for a fix specific to the vulnerability, the probability that a user has applied a vendor-supplied patch that fixes the problem is zero, so the exploit would remain available. Zero-day attacks are a severe threat.

Source: https://en.wikipedia.org/wiki/Zero-day_(computing)

The Art of Exploitation

- Crafting exploits requires a lot of domain specific knowledge
- One domain is memory corruption vulnerabilities in software to hijack the control flow and make it do whatever you want it to do



More info on "**What is a "good" memory corruption vulnerability?**"

- https://googleprojectzero.blogspot.ch/2015/06/what-is-good-memory-corruption.html

Memory corruption vulnerabilities

- Buffer overflow
  - No bounds-checking
  - Erroneous bounds-checking

```
char target[128];
strcpy(target, source)
```

- Indexing errors
  - No arbitrary control
  - Can "jump" over memory areas, e.g., stack canary

```
for (k = 0; k < num_nonzero; k+=1) {
   i += RLCoeffs[k*2+0];
   temp[i] = RLCoeffs[k*2+1];
   i += 1;
}
```

- Arbitrary writes
  - No or erroneous bound-checks

```
array[user_sup_index] = user_sup_data
```

- Use-After-Free
  - Memory that was freed and ev. reused

```
char* ptr = (char*)malloc (SIZE);
if (err) {
   abrt = 1;
   free(ptr);
}
…
if (abrt) {
   logError("operation aborted", ptr);
}
```

- Type confusion

Indexing errors:

Input data controlled by the user can influence the index when writing into an array. Non-sequential indexing might even allow jumping over stack-canaries and to overwrite the return address without being detected. An easy-to-understand example of a real-world vulnerability can be found here:
https://bugs.chromium.org/p/project-zero/issues/detail?id=291&redir=1

Arbitrary writes:

Input data controlled by the user can control the index in an arbitrary way. An easy-to-understand example of a real-world vulnerability can be found here:
https://marcograss.github.io/security/android/cve/2016/05/03/cve-2016-2443-msm-kernel-arbitrary-write.html

Use-After-Free:

According to the Use After Free definition on the Common Weakness Enumeration (CWE) website, a Use After Free scenario can occur when "the memory in question is allocated to another pointer validly at some point after it has been freed. The original pointer to the freed memory is used again and points to somewhere within the new allocation. As the data is changed, it corrupts the validly used memory; this induces undefined behavior in the process."

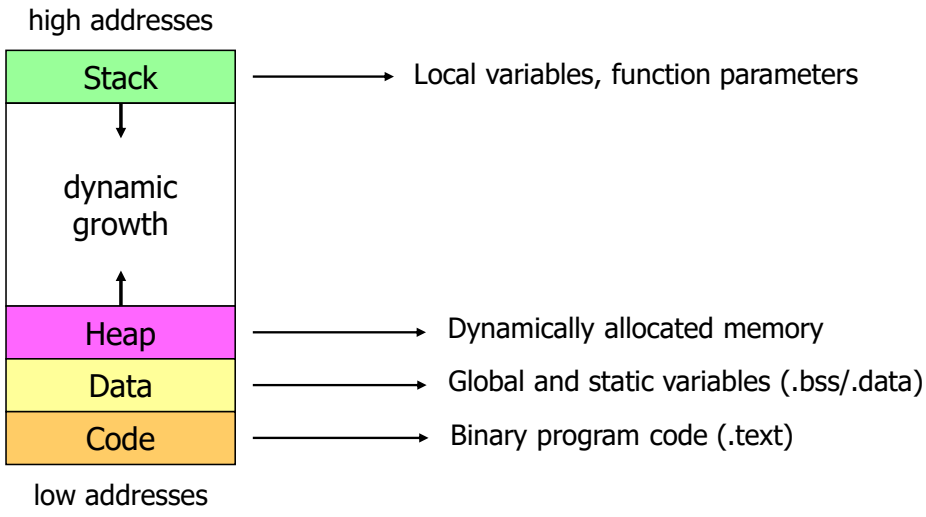
Hence, Use-After-Free specifically refers to the attempt to access memory after it has been freed, which can cause a program to crash or, in the case of a Use-After-Free flaw, can potentially result in the execution of arbitrary code or even enable full remote code execution capabilities.

A writeup on a real-world example can be found here:
https://scarybeastsecurity.blogspot.ch/2013/02/exploiting-64-bit-linux-like-boss.html

Type confusion:

When triggering a type confusion vulnerability, a piece of code has a reference to an object which it believes to be of type A (the API type), but really it is confused and the object is of type B (the in-memory type). Depending on the in-memory structure of type A vs. type B, very weird but usually fully deterministic side-effects can occur. A real-world example can be found here:
https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-kernel-exploit/

Typical Memory Organisation



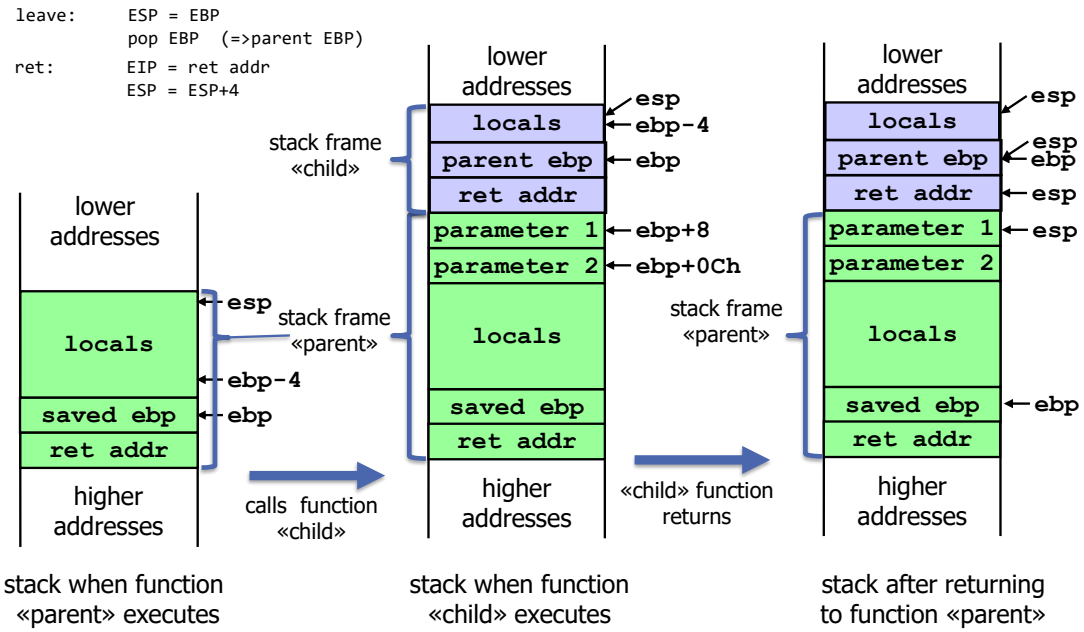
- Every process runs in its own virtual address space
- The address space of a process can be separated into segments

Note I: Buffers are usually either on the heap or the stack. For the discussion of the concept of return-oriented programming and how to make use of it, leveraging buffer overflows of buffers on the stack is much easier. The main reason for this is that the heap-layout at a certain point in the code is in general much less predictable than the stack.

Memory layout of a Process

- The **Code** segment (often also called .text segment) contains the executable program (in machine language). This is where the instruction pointers should normally point to.
- The **Data** segment (often also called .bss/.data segment) contains all data, that exist exactly once during program executions (static and global variables).
- The **Stack** segment contains data, that are produced and destroyed dynamically during run-time in a “well-defined order”. This includes function arguments and local variables.
- The **Heap** segment is used for all other, dynamically during run-time generated data (malloc in C, new in Java)

Stack Layout and Function Calls - Revisited



There are different ways to organize and use the stack. For example, parameters might be passed on the stack or in CPU registers or frame pointers (EBP) might be omitted.

Here, we look at the stack of x86 (32-bit) binaries compiled with GCC with the **cdecl** calling convention (pronounced “see-deck-'ll”, rhymes with "heckle"). It is the default calling convention for x86 C compilers, although many compilers provide options to change the calling conventions used. To locate the top and bottom of the current stack frame, the two pointers (CPU registers) ESP and EBP are the key. The stack pointer (ESP) point always to the topmost entry on the stack. It is used to indicate where data shall be pushed onto the stack or popped from the top of the stack. The base pointer (EBP) always points to the bottom of the currently used stack frame (not exactly the bottom, but one position above the return address).

Function call:

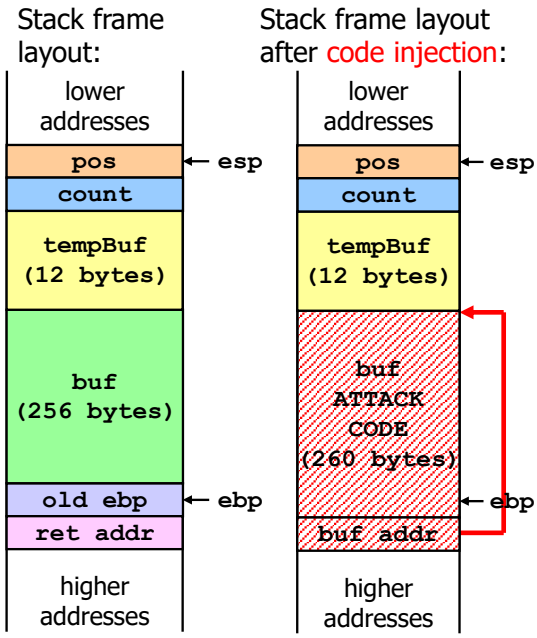
If the function to be called has parameters, they are placed on the stack. The parameters belong to the stack frame of the caller (parent).

1. A call instruction is executed which leads to the creation of a new stack frame:
   - It pushes the content of the EIP register onto the stack (ret addr). Note that the push instruction adjusts the stack-pointer accordingly.
   - The address provided to the call instruction is loaded into EIP and execution continues from there.
2. The function **prologue** of the called function is executed. It consists of a few lines of code at the start of the function that prepare the stack and registers for use within the function.
   - It pushes the EBP onto the stack to save the EBP of the calling function.
   - The current value of ESP is stored in EBP – it marks the base of the new stack frame
   - Space for local variables is allocated (if any) by altering the ESP accordingly.
3. The function performs its work.
4. The **epilogue** of the function is executed – it restores the stack and registers to their state before the function was called. The epilogue often contains the leave instruction followed by a ret instruction. If the leave instruction is not used, there should be multiple other instructions with the same result.
   - `leave` sets the ESP to EBP so that it points to the *saved EBP* and then it restores the EBP from the parent function by popping the *saved EBP* from the stack. Hence, we end up with ESP pointing to the return address.
   - `ret` pops the return address off the stack and into EIP so that the program continues execution from just after where the original call was made. After popping the return address, ESP points to the top of

the stack of the stack frame of the parent function.

Stack-Based Buffer Overflow - Revisited

```
char buffer[256], tmpBuf[12];
int count = 0, pos = 0;

// Read data from socket,
// and copy it into buffer
cnt = recv(sock,tmpBuf,12,0);
while (cnt > 0) {
  memcpy(buffer+pos,tmpBuf,cnt)
  pos += cnt;
  cnt = recv(sock, tmpBuf,12,0);
}
```



Protections - Revisited

- ASLR – Address Space Randomization
  - Places some or all memory segments (stack, heap, code, data,…) at random positions in the virtual address space
  - Effectiveness (address entropy) depends on placement algorithm and architecture (32-bit much lower than on 64-bit)
- Non-executable memory:
  - Marks memory as non-executable at the page level (4 kB or greater)
  - Naming: NX (Linux), DEP (Windows), XN (ARM)
    - XN = eXecute Never, NX = No-eXecute, DEP= Data Execution Prevention
- Stack canaries
  - Protect parts of the stack (e.g., ret address) from sequential overwrites

We need more sophisticated technique than the simple buffer overflow technique from before!

For more details on **ASLR** and **non-executable memory**, the wikipedia articles are a good starting point. However, keep in mind that if implementation details are mentioned, these might be version specific and change over time.

https://en.wikipedia.org/wiki/Address_space_layout_randomization

https://en.wikipedia.org/wiki/Executable_space_protection

All modern general-purpose architectures/CPUs have hardware support for **non-executable memory**.

On x86 starting with Pentium 4 or later (64-bit mode or 32-bit with the physical address extension page-table format)

A good blog post on **stack canaries** and their implementation in GCC:

https://access.redhat.com/blogs/766093/posts/3548631

Return Oriented Programming (ROP)

ROP – Overview

- ROP is a technique that allows to execute arbitrary code on the target without submitting/injecting any code (first published in 09/2005 by Sebastian Krahmer)
- Instead, ROP makes use of the code of the application itself
- Pieces of code that are useful for ROP are called gadgets
- An important property of gadgets is that at the end of a gadget, there is code to pass control to another gadget (the attacker defines to which one)
- ROP is a control-flow hijacking attack – by altering data relevant for the control-flow it makes a new program out of an existing one
  - There is enough code in most applications to implement any functionality
- NX/DEP protection is completely useless against ROP-based remote code execution attacks

ROP by Example – Calling Functions (1)

- Let us assume we have a program with a buffer-overflow vulnerability
- We want to make a call to an EXISTING function – the target function – so that NX/DEP protection mechanisms are not triggered
- For now, we assume that
  - we know the address of the target function
  - no stack canaries are used and/or we know their values

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

void target() {
  printf("target\n");
}

void process_arg(char* arg) {
  char buf [1024];
  strcpy(buf, arg);
}

int main( int argc, char** argv) {
  if(argc != 2) {
    system("echo More #args!");
    return 2;
  } else {
    process_arg(argv[1]);
  }
  return 0;
}
```

ROP by Example – Calling Functions (2)

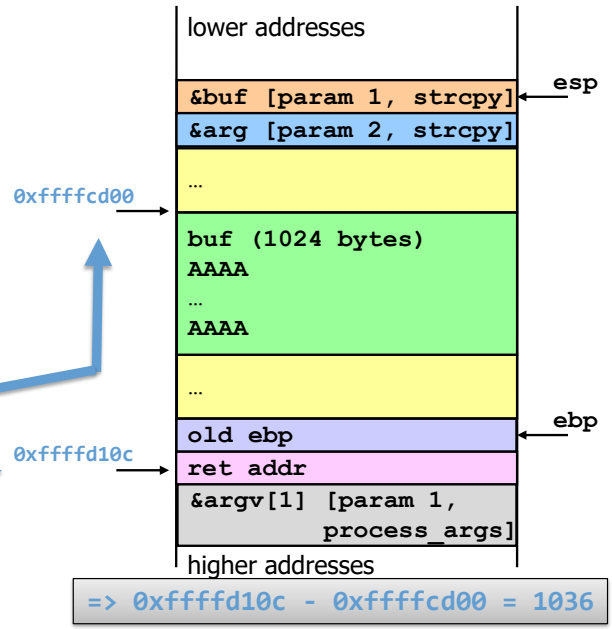
- How to call the target function?

- Basic idea:
  - Find address of target
  - Determine number of bytes required to overwrite return address
  - Craft an input so that the return address is overwritten with the address of target

ROP by Example – Calling Functions (3)

```
gdb> print target
$1 = {<text variable, no debug info>} 0x8049182
<target>

gdb> disass process_arg
…
0x080491d0 <+35>: call 0x8049030 <strcpy@plt>
0x080491d5 <+40>: add  esp,0x10
…

gdb> break *0x080484be
…

gdb> run `p2bin 'b"A" * 1024'`
…

gdb> context stack
…
gdb> info frame
…
 Saved registers:
  ebx at 0xffffd104, ebp at 0xffffd108,
  eip at 0xffffd10c
```

Stack frame layout:



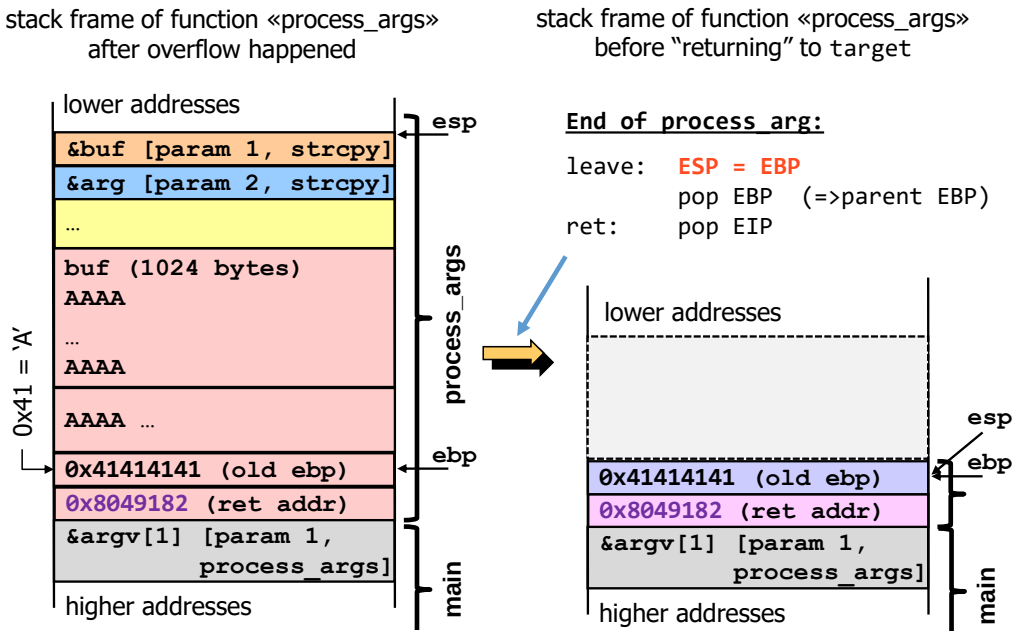
=> 0xffffd10c - 0xffffcd00 = 1036

ROP by Example – Calling Functions (4)

- How to call the target function?
- Basic idea:
  1. Find address of target
     => 0x8049182
  2. Determine number of bytes required to overwrite return address
     => 1036
  3. Craft an input so that the return address is overwritten with the address of target
     `./rop-demo `p2bin 'b"A"*1036 + pack("<L“, 0x08049182)’`
  4. Result:
     target
     [2]   2467 segmentation fault ./rop `python -c 'print …`

Why does this happen?

ROP by Example – Calling Functions (5)



If our overflow did not disrupt the execution of the process_args function, we reach the end of the function.

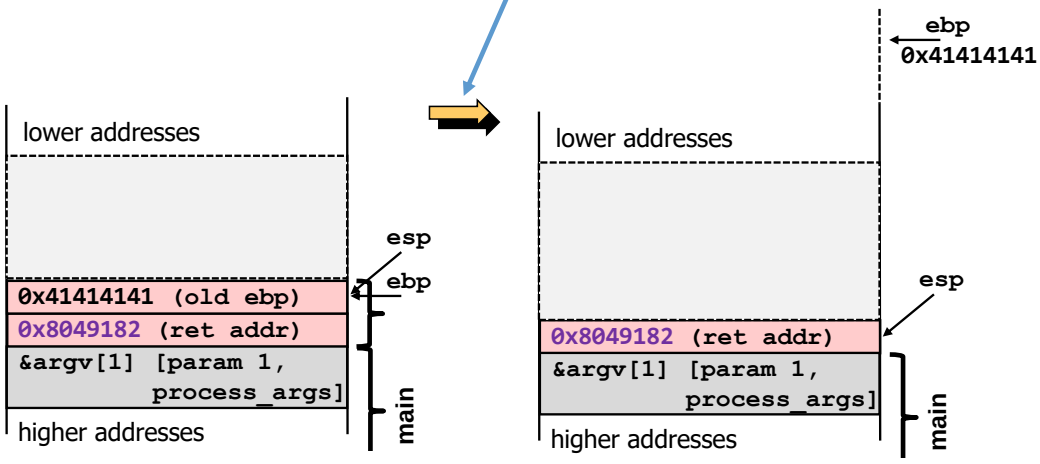
At the end of the function, the stack looks like shown on the right:

- The top of the stack (esp) now points to the saved base pointer of the previous function. Since we overwrote it with «A»s, this base pointer does not point to the base of the stack frame of the main function anymore.
- The return address points to the target function. Hence, a ret instruction consuming it would start to execute this function

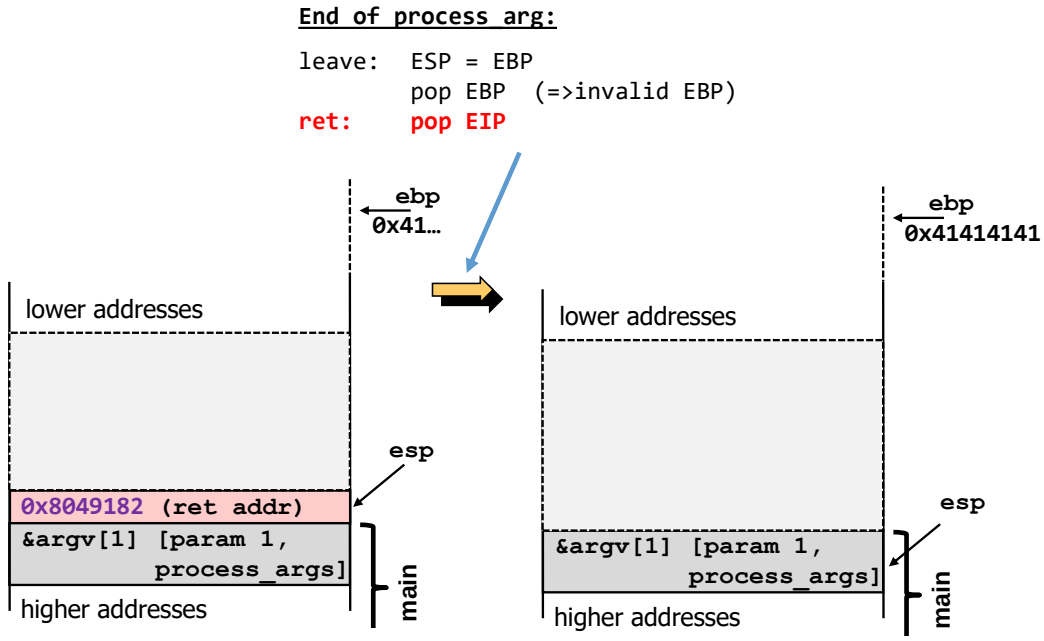
ROP by Example – Calling Functions (6)

End of process_arg:

```
leave:  ESP = EBP
        pop EBP  (=>invalid EBP)
ret:    pop EIP
```



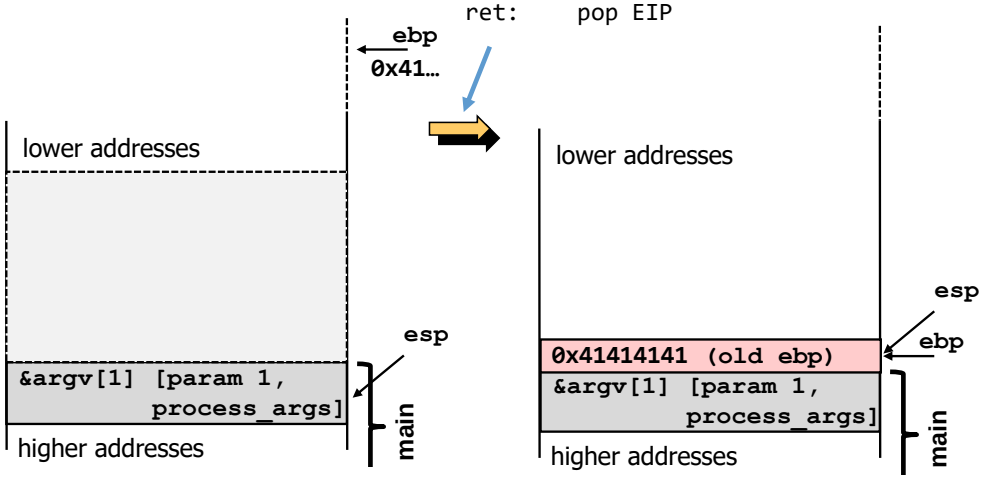
ROP by Example – Calling Functions (7)



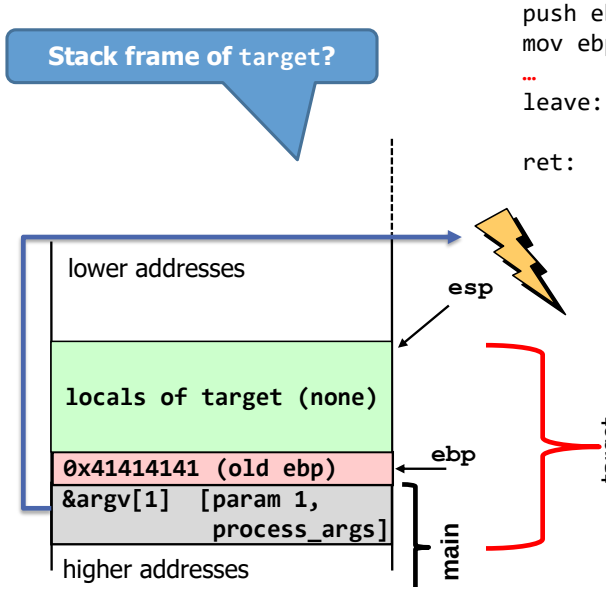
ROP by Example – Calling Functions (8)

Prolog and epilog of target:

```
push ebp        //save base pointer
mov ebp, esp    //set ebp to esp
…
leave:  ESP = EBP
        pop EBP  (=>invalid EBP)
ret:    pop EIP
```



ROP by Example – Calling Functions (9)



Prolog and epilog of target:

```
push ebp          //save base pointer
mov ebp, esp      //set ebp to esp
…
leave:  ESP = EBP
        pop EBP  (=>invalid EBP)
ret:    pop EIP
```

Here, there should be the **return address** to which the control flow returns when the target function ends (always below saved ebp).

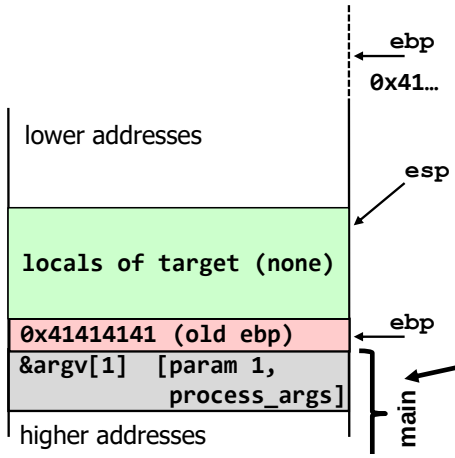
Why is there no return address?

- We did not go to the `target` function with a `call` instruction but with a ret instruction.
- The `call` instruction would put the instruction after the `call` instruction as return address on the stack.

ROP by Example – Calling Functions (10)

Prolog and epilog of target:

```
push ebp        //save base pointer
mov ebp, esp    //set ebp to esp
…
leave:  ESP = EBP
        pop EBP  (=>invalid EBP)
ret:    pop EIP
```



When executing `ret`, we have a segmentation fault - the address of argv[1] is not pointing to a code segment.

ROP by Example – Calling Functions (11)

- To "fix" the problem, we could use the buffer overflow to overflow beyond the return address and write several addresses of functions, one after each other
- What would happen, if we execute the following command?

```
./rop `p2bin 'b"A"*1036 +
        pack("<L", 0x08049182) +
        pack("<L", 0x08049182) +
        pack("<L", 0x08049182)'`
```

Result:

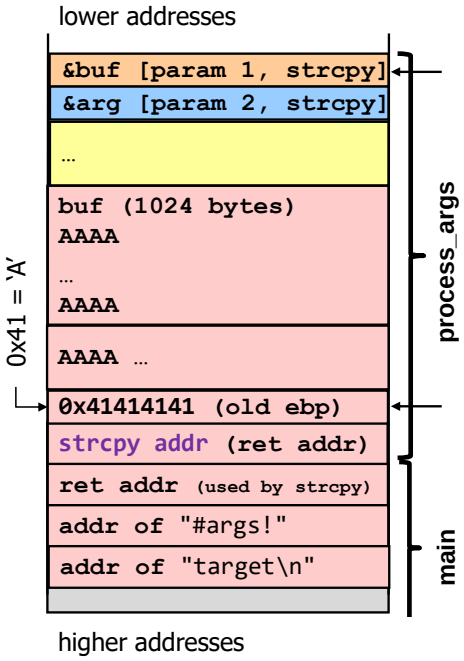
```
target
target
target
[1] 1276 segmentation fault  ./rop
```

Great! We can chain functions and create new "programs" by doing so. Right?

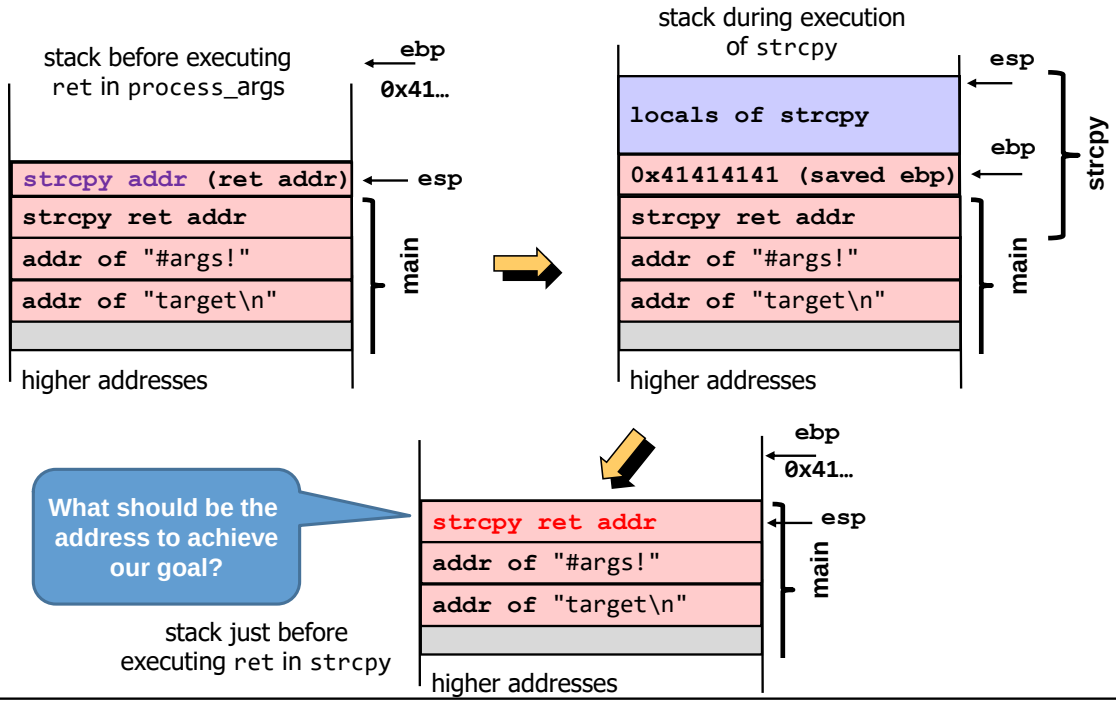
How about calling functions that have parameters?

ROP by Example – Calling Functions with Parameters (1)

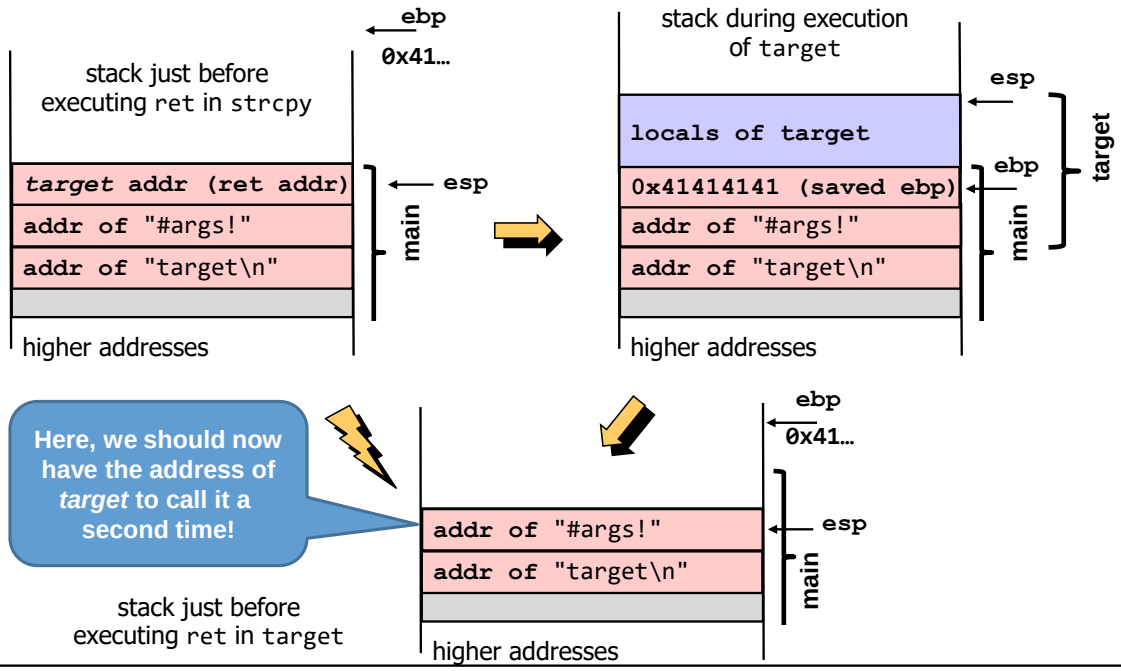
- Let's now try to craft an input that:
  - Calls `strcpy` and replaces the string `"target\n" with "#args!"`
  - Call the target function twice
- To complete the first step:
  - The return address must be overwritten with the address of `strcpy`
  - It must be followed by the return address that `strcpy` should use when it returns
  - It must be followed by the two arguments required by `strcpy` [=two memory addresses]



ROP by Example – Calling Functions with Parameters (2)



ROP by Example – Calling Functions with Parameters (3)



ROP by Example – Gadgets to the Rescue

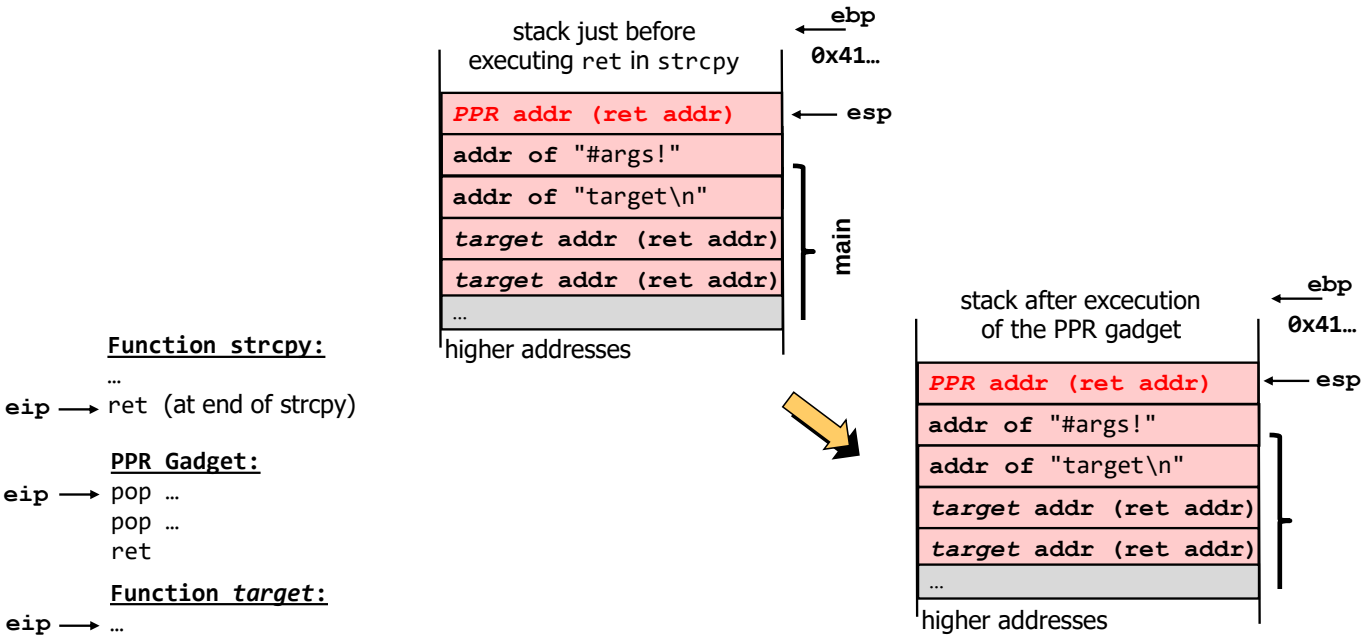
- If we "return" to a function that has parameters, we end up with no valid return address on top of the stack when the function to which this function returns, itself returns => segmentation fault
- A specific position on the stack can't contain a valid return address and an argument at the same time
- Instead of "returning" to the next function directly, we return to a gadget that adjusts the stack so that the chain does not break

- Basic idea – Jump to a gadget that:
  1. removes the arguments of the function that was executed before
  2. executes a `ret` instruction

  => In the example, we should look for a `pop; pop; ret;` gadget (PPR):

```
pop …
pop …
ret
```

ROP by Example – Calling Functions with Parameters (3)



ROP by Example – Calling Functions with Parameters (4)

- Hence, our input must look as follows:

```
./rop `p2bin 'b"A" * 1036 +
       <addr strcpy> +
       <addr PPR> +
       <addr "#args!"> +
       <addr "target\n"> +
       <addr target> +
       <addr target>'`
```

ROP – Generic Pattern

```
Before   After
LOCALS   <FILLER>       <- Filler bytes from overflow to reach EIP
EIP      FUNCT_1        <- EIP overwritten with FUNCT_1's address
…        PPR            <- Return address that FUNC_1 will use
                           Points to a pop; pop; ret; gadget
         ARG1_FUNC_1    <- Argument for FUNCT_1
         ARG2_FUNC_1    <- Argument for FUNCT_1
         FUNC_2         <- Function to be called after FUNC_1
         PR             <- Return address that FUNC_1 will use
                           Points to a pop; ret; gadget
         ARG1_FUNC_2    <- Argument for FUNCT_2
         FUNC_3         <- Function to be called after FUNC_2
         …
```

- For stack-based ROP, we must prepare the stack with a sequence of addresses to existing pieces of code
- If we call functions taking parameters, we must use gadgets to remove the parameters after the call

ROP – Is ROP dying?

- For ROP to work, we must know the addresses of gadgets
- All modern general purpose operating systems have support for ASLR
- If ASLR is used for **all** code, the position of at least **one** gadget must be acquired using an information leak or brute force
  - On 32-bit systems, entropy is often low enough so that brute force is an option (=>see exploitation lab)
- Why just one?
  - The code segment of a binary is put at a random location, the code within remains the same
  - If we have a copy of the binary, we can inspect the code segment and calculate the address of all other gadgets based on the known one
- There are some defenses that may complicate things like:
  - Control Flow Integrity in the Clang compiler frontend (Linux and Windows)

More details on Control Flow Integrity in Clang:
https://clang.llvm.org/docs/ControlFlowIntegrity.html

ASLR (and PIC/PIE) Support

- In most modern operating systems (security critical) libraries/binaries are compiled as PIE and often, it's the default with modern compilers
  - ASLR != ASLR:
  - An example for how complicated the situation is (for Windows)
    - https://www.fireeye.com/blog/threat-research/2020/03/six-facts-about-address-space-layout-randomization-on-windows.html
  - Compiler default YES does not imply that all programs are compiled with it

* Embedded systems and IoT devices in general often lack features like DEP or ASLR or have them turned off because of performance considerations

	OS binaries	Compiler Default
Android 5.0+	ENFORCED	ENFORCED
RHL 8+ / Fedora 24+	YES	YES
Ubuntu 16.10+	ENFORCED (64-bit) package-list (32-bit)	YES
Windows 10	YES	YES (for VS 2012+)
Cisco ASA* ASA: Adaptive Security Appliance	NO	NO

In computing, **position-independent code (PIC)** or position-independent executable (PIE) is a body of machine code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address. PIC is commonly used for shared libraries, so that the same library code can be loaded in a location in each program address space where it will not overlap any other uses of memory (for example, other shared libraries). PIC was also used on older computer systems lacking an MMU, so that the operating system could keep applications away from each other even within the single address space of an MMU-less system.

Position-independent executables (PIE) are executable binaries made entirely from position-independent code.

Source: https://en.wikipedia.org/wiki/Position-independent_code

Linux ASLR/PIE support:

ASLR/PIE support details for RHEL, Fedora and Ubuntu can be found here:

- RHEL 7 and Fedora 24: https://fedoraproject.org/wiki/Security_Features_Matrix
- RHEL 8: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/8.0_release_notes/index
- Ubuntu / Debian: https://wiki.ubuntu.com/Security/Features/

Windows and ASLR:

ASLR was introduced in Windows Vista and has been included in all subsequent releases of Windows. As with DEP, ASLR is only enabled by default for core operating system binaries and applications that are explicitly configured to use it via a new linker switch.

https://www.microsoft.com/security/sir/strategy/default.aspx#!section_3_3

Hence, by default, Windows 10 and lower have ***some*** DEP/ASRL protection but not a system-

wide protection. However, with **Windows 10**, ASLR and DEP can be enforced system wide. The problem here is that this might NOT be compatible with some applications! In this case, and for more fine-tuning and special protections, you have to use the end of life tool EMET.

- System-wide DEP can be configured using the BCDEdit utility. Microsoft indicates, "*Before setting BCDEdit options you might need to disable or suspend BitLocker and Secure Boot on the computer."* To change the DEP setting to AlwaysOn, in a CMD prompt with administrative privileges run:
  bcdedit.exe /set {current} nx AlwaysOn
- System-wide ASLR can be configured by specifying the following registry value:
  [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management]
  "MoveImages"=dword:ffffffff

Windows – Relocatable Code/Module:

A relocatable module (exe or dll) doesn't necessarily need to have ASLR enabled but a module that has ASLR enabled needs to be relocatable.

A module that is ASLR-enabled (using the /DYNAMICBASE linker switch) will be loaded at a random address regardless of its ImageBase (the preferred load address) and therefore it must be relocatable or it cannot be loaded.

If a module is not ASLR-enabled the loader will first try to load it at the ImageBase. If that is not possible (the memory is already allocated), it will try to load it at another address; if the module is relocatable it will succeed, if not it will fail.

How to identify a relocatable module?

A module that is not relocatable will have the IMAGE_FILE_RELOCS_STRIPPED (0x0001) bit flag set in the Characteristics field of their File Header. A relocatable module will have this bit cleared and it will also have a section with relocations (like .reloc). You can examine that flag with software like PEView or dumpbin /headers your_module.exe (or dll)

How to identify a module with ASLR enabled?

An ASLR-enabled module will be relocatable (the relocs stripped flag unset) and it will also have the IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE (0x0040) flag set in the DllCharacteristics field of the optional header. The DllCharacteristics is used for both EXEs and DLLs regardless of its name.

Again you can check for the presence of this flag set with a PE file explorer like PEView or with dumpbin /headers your_module.dll (or exe).

And what about Stack Canaries?

- They prevent us from using the return address for control-flow hijacking
- To start our ROP-Chain, we need a different entry point
  - Overwrite a function pointer on the stack
  - Use a vulnerability that allows us to «jump over» the canary
  - Use a vulnerability that leaks information so that we know the canary
  - Overwrite a structured exception handler (SHE, Windows only)
- Overwrite a function pointer on the stack:
  - Point it to a ROP gadget that adjusts the ESP (top of the stack) to point to the top of the user-controlled buffer
  - The `ret` of the gadget will then take the first value in the user-controlled buffer as the return address and therewith start executing the chain

Is ROP Dying?

- To bypass «full» ASLR, attackers must be able to read and leak memory content
  - With the RCE vulnerability or an additional information leakage-only vulnerability
- If you can read the memory and leak information to the attacker, in many cases, you can also write into it => No ROP required
  - «Write Once, Pwn Anywhere» (Blackhat USA, 2014); Trigger a vulnerability giving programmatic read/write of virtual memory to overwrite critical data to gain code execution
- There are many exploits that do not use ROP anymore:
  - Corruption of «safemode» flag of Jscript.dll to enable use of Wscript.Shell COM method => execute shell commands (CVE-2014-6332)
  - Vulnerability in Microsoft Silverlight ScriptObject discovered and used by Vitaly Toropov to bypass DEP without needing ROP (MS13-022)
  - Create legitimate code in non-NX memory using JIT engines (CVE-2016-0034)
  - Many more techniques exist publicly and many unknown ones too

https://www.corelan.be/index.php/security/corelan-ropdb/

Summary

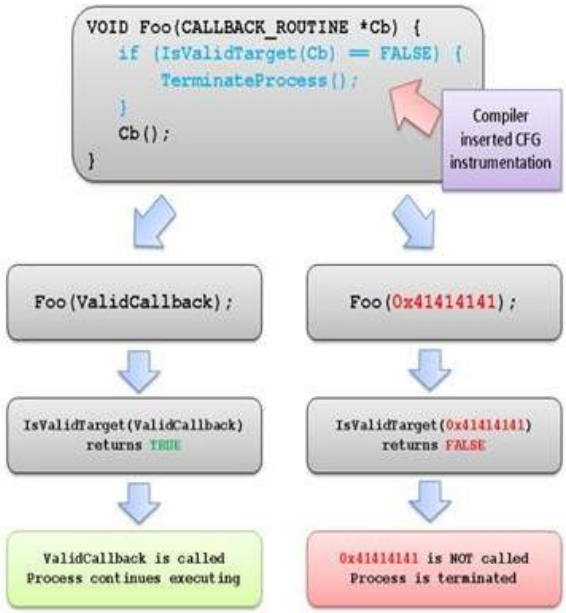
- ROP is a powerful concept to circumvent NX/DEP protections
- ROP implements programs by hijacking the control flow of the program and by chaining piece of code (gadgets) already present in the program
- It does NOT solve the stack canary or ASLR problem
- Protections like ASLR or the rather new control flow integrity in Windows 10 makes ROP attacks more difficult
- Importance of ROP is decreasing in attacks versus general purpose computing systems => what about the IoT world?

More and deeper? Very good article:
https://www.cyberark.com/resources/threat-research-blog/a-modern-exploration-of-windows-memory-corruption-exploits-part-i-stack-overflows

Appendix

Control Flow Guard

- Supported in Windows 10 and Windows 8.1 with (KB3000850)
- Applications must be compiled and linked with this feature
  - `/guard:cf` compiler option
- Detects invalid indirect calls
- Protects against function pointer overwrites
- Does not protect against vulnerabilities triggering a valid code path that would be invalid for the provided data
  - E.g., «admin» path instead of «user» path as normal user



More Info can be found here:
https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx