Workshop ASI

Get Fun with Buffer Overflows

Mario Alviano

University of Calabria, Italy

23 June 2019, 9:30–12:30, Aula Seminari DIMEG

Outline

Outline

1. Introduction
   - Context and goal
   - Overview and example
2. Assembly
   - Warm up
   - Computer architecture
   - Most frequent instructions
   - C/C++ calling convention
3. Buffer overflow and shellcode
   - Simple examples
   - Privilege escalation
   - Help yourself with peda
   - Return-to-libc and ROP
4. Conclusion

Outline

Search for “buffer overflow” on CVE

Common Vulnerabilities and Exposures

CVE List | CNAs | Board | About | News & Blog

NVD
Go to for:
CVSS Scores
CPE Info
Advanced Search

Search CVE List | **Download CVE** | **Data Feeds** | **Request CVE IDs** | **Update a CVE Entry**

TOTAL CVE Entries: 116418

HOME > CVE > SEARCH RESULTS

Search Results

There are **9940** CVE entries that match your search.

Name	Description
CVE-2019-9956	In ImageMagick 7.0.8-35 Q16, there is a stack-based buffer overflow in the function PopHexPixel of coders/ps.c, which allows an attacker to cause a denial of service or code execution via a crafted image file.
CVE-2019-9928	GStreamer before 1.16.0 has a heap-based buffer overflow in the RTSP connection parser via a crafted response from a server, potentially allowing remote code execution.
CVE-2019-9895	In PuTTY versions before 0.71 on Unix, a remotely triggerable buffer overflow exists in any kind of server-to-client forwarding.
CVE-2019-9810	Incorrect alias information in IonMonkey JIT compiler for Array.prototype.slice method may lead to missing bounds check and a buffer overflow. This vulnerability affects Firefox < 66.0.1, Firefox ESR < 60.6.1, and Thunderbird < 60.6.1.
CVE-2019-9773	An issue was discovered in GNU LibreDWG 0.7 and 0.7.1645. There is a heap-based buffer overflow in the function dwg_decode_eed_data at decode.c for the z dimension.
CVE-2019-9770	An issue was discovered in GNU LibreDWG 0.7 and 0.7.1645. There is a heap-based buffer overflow in the function dwg_decode_eed_data at decode.c for the y dimension.
CVE-2019-9767	Stack-based buffer overflow in Free MP3 CD Ripper 2.6, when converting a file, allows user-assisted remote attackers to execute arbitrary code via a crafted .wma file.
CVE-2019-9766	Stack-based buffer overflow in Free MP3 CD Ripper 2.6, when converting a file, allows user-assisted remote attackers to execute arbitrary code via a crafted .mp3 file.
CVE-2019-9687	PoDoFo 0.9.6 has a heap-based buffer overflow in PdfString::ConvertUTF16toUTF8 in base/PdfString.cpp.

An ever green

- So many vulnerabilities related to buffer overflow
- Severe consequences (especially for C/C++ programs)
    - Denial of service
    - Remote code execution
    - Privilege escalation

An ever green

- So many vulnerabilities related to buffer overflow
- Severe consequences (especially for C/C++ programs)
  - Denial of service
  - Remote code execution
  - Privilege escalation

Who cares about C/C++

You should! Your OS is written in C/C++. Your browser too.

Goal of this lecture

- Understand low level mechanisms of program execution
- Exploit common mistakes to deviate from standard behavior
- Craft and inject shellcodes
- Practice with `gdb` and `peda`

We work with linux x86 (32-bits)

Just because it is simpler than other (64-bits) OSes for this purpose

Outline

Program execution

- The CPU executes machine instructions
- A register stores the instruction pointer (IP)
- Conditional instructions are used to break sequentiality

Program execution

- The CPU executes machine instructions
- A register stores the instruction pointer (IP)
- Conditional instructions are used to break sequentiality
- The control section of the memory stores instructions
- The data section contains program's data

Program execution

- The CPU executes machine instructions
- A register stores the instruction pointer (IP)
- Conditional instructions are used to break sequentiality
- The control section of the memory stores instructions
- The data section contains program's data

Such a separation is not always checked

Procedures

- Programs are often split in procedures

Procedures

- Programs are often split in procedures
- Calling a procedure requires to modify IP
- After the called procedure terminates, IP must return to the callee procedure

Procedures

- Programs are often split in procedures
- Calling a procedure requires to modify IP
- After the called procedure terminates, IP must return to the callee procedure
- This is achieved by storing the return address in the stack
- Buffers local to a procedure are also stored in the stack

Procedures

- Programs are often split in procedures
- Calling a procedure requires to modify IP
- After the called procedure terminates, IP must return to the callee procedure
- This is achieved by storing the return address in the stack
- Buffers local to a procedure are also stored in the stack

A buffer overflow may replace the return address

The Morris Worm

The `finger` program

Provides information on user@machine

The Morris Worm

The `finger` program

Provides information on user@machine

It had a buffer overflow

- Assumed that people would rely on short names
- Allocated only 11 bytes for user@machine (plus null character)
- Morris provided a long string to execute a shellcode

The Morris Worm

The `finger` program

Provides information on user@machine

It had a buffer overflow

- Assumed that people would rely on short names
- Allocated only 11 bytes for user@machine (plus null character)
- Morris provided a long string to execute a shellcode

Why this attack was possible

- Separation of data and instructions was not checked
- `finger` ran with root privilege

Outline

Outline

What is assembly language?

- The CPU manages arithmetical, logical, and control activities
- The CPU follows machine language instructions
- Machine language instructions are strings in $\{0, 1\}^*$
- Assembly is almost one-to-one to machine language

Why studying an assembly language?

To understand the following:

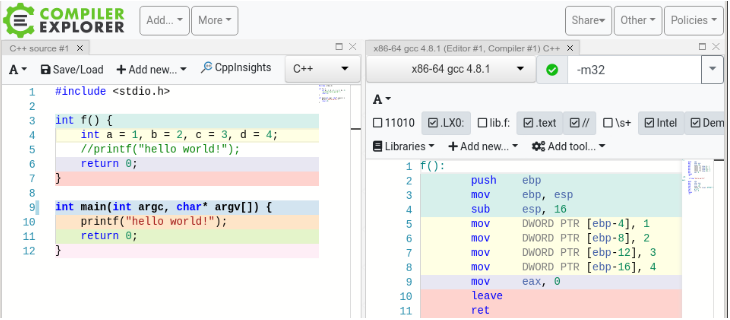
- How programs interface with OS, processor, and BIOS
- How data is represented in memory and other external devices
- How the processor accesses and executes instruction
- How instructions access and process data
- How a program accesses external devices

Setup

We are going to use an online disassembler

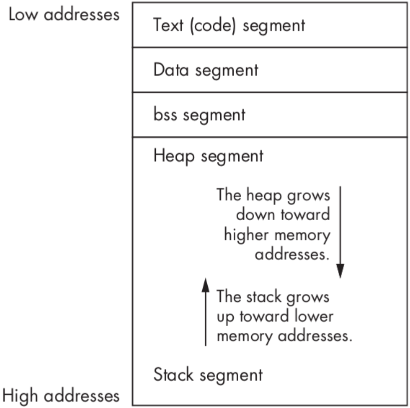
`https://godbolt.org/`

- Select C++ as language (on the left)
- Select x86-64 gcc 4.8.1 as compiler (on the right)
- Set `-m32` as command-line option



Outline

Memory segments

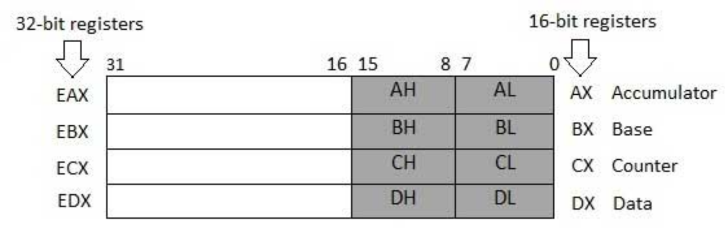


- Text: assembly code
- Data: global initialized variables
- BSS: global unitialized variables
- Heap: dynamically allocated memory
- Stack: local (and temporary) memory

Registers of an x86 processor

- General registers
  - Data registers
  - Pointer registers
  - Index registers
- Control registers
- and others

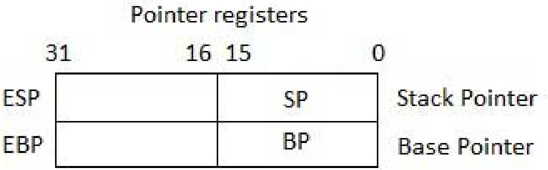
Data registers



- Four 32-bit data registers
- Used for arithmetic, logical and other operations
- Can be also used as 16-bit or 8-bit data registers

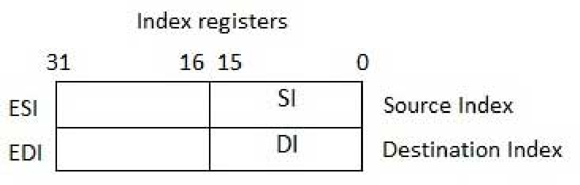
AX, BX, CX, DX use bits 0-15

Pointer registers



- Three 32-bit pointer registers
  - ESP: address of current top stack element
  - EBP: address of the stack frame
- Can be also used as 16-bit pointer registers

Index registers



- Two 32-bit index registers
- Used for addressing memory
- Can be also used as 16-bit pointer registers

Control registers

- EIP: 32-bit instruction pointer register
  - Address of the next instruction to be executed

Flag:					O	D	I	T	S	Z		A		P		C
Bit no:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- EFLAGS: 32-bit flags register
  - Overflow Flag (OF): 1 if the last arith. op. overflowed
  - Direction Flag (DF): left-to-right (0) or right-to-left (1) processing of strings
  - Interrupt Flag (IF): ignore (0) or process (1) external interrupts
  - Trap Flag (TF): 1 for single-step execution (to debug)
  - Sign Flag (SF): 0 if the last arith. op. gave a positive result
  - Zero Flag (ZF): 1 if the last arith. op. gave 0
  - Auxiliary Carry Flag (AF): the carry from bit 3 to bit 4 in the last arith. op.
  - Parity Flag (PF): parity bit of the last arith. op.
  - Carry Flag (CF): the carry of the high-order bit in the last arith. op.

Outline

Arithmetic instructions

- `inc destination`
- `dec destination`
- `add destination, source`
- `sub destination, source`

At least one operand must be different from memory address

Logical instructions

Bitwise logical operations, storing the result in `operand1`:

- `and operand1, operand2`
- `or operand1, operand2`
- `xor operand1, operand2`
- `not operand1`

Bitwise AND, just setting flags (ZF is set to 1 if the AND is 0)

- `test operand1, operand2`

Unconditional jump

- `jmp label`
  Set IP to the address of the given label

```
MOV  AX, 00    ; Initializing AX to 0
MOV  BX, 00    ; Initializing BX to 0
MOV  CX, 01    ; Initializing CX to 1
L20:
ADD  AX, 01    ; Increment AX
ADD  BX, AX    ; Add AX to BX
SHL  CX, 1     ; shift left CX, this in turn doubles the CX value
JMP  L20       ; repeats the statements
```

Conditional jump

- `j<condition> label`

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/Equal or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

- Often preceded by `cmp operand1, operand2`
- It is like `sub`, but `operand1` is not changed
- Only flags are affected

```
INC     EDX
CMP     EDX, 10 ; Compares whether the counter has reached 10
JLE     LP1     ; If it is less than or equal to 10, then jump to LP1
```

Let's try a few examples

- Go to `https://godbolt.org/` and try common constructs
- Colors help to understand how C/C++ is compiled into assembly instructions
- Assign unique constants to variables to easily identify them
- Try if..then, if..then..else, while, do..while, for

Outline

Stack instructions

- `push operand`
- `pop address/register`
- Used for local variables
- Used to create cached copies
- Used for passing arguments to procedures

Subroutines

- Subroutines are identified by labels
- Subroutines are called by `call label`
  - Pushes EIP into the stack, and jumps to `label`
- Each subroutine terminates with `ret`
  - Pops an address from the stack, and jumps to it

Calling convention

- How to share subroutines?
- We must agree on some strategy to pass paramenters
- Several conventions do exist
- We will consider the C/C++ convention
- Essentially, use the stack!

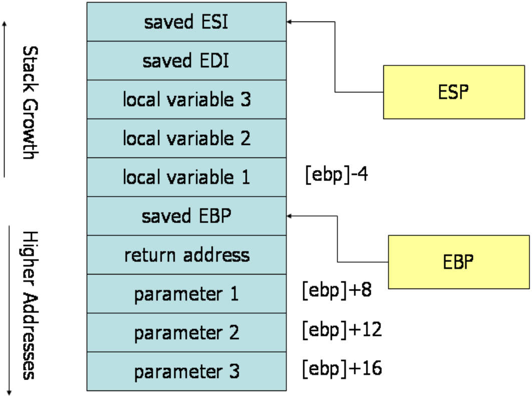
Calling convention

- How to share subroutines?
- We must agree on some strategy to pass paramenters
- Several conventions do exist
- We will consider the C/C++ convention
- Essentially, use the stack!

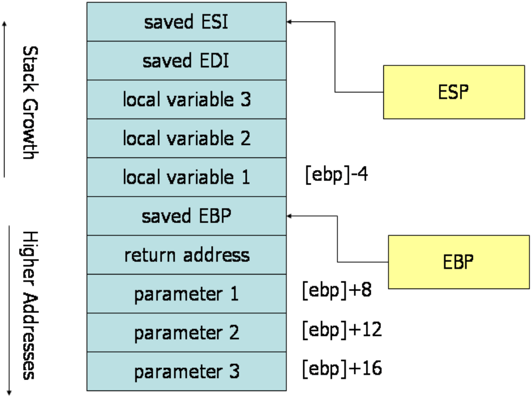
Two sets of rules

1. The first set is for the caller
2. The second set is for the callee

Caller rules

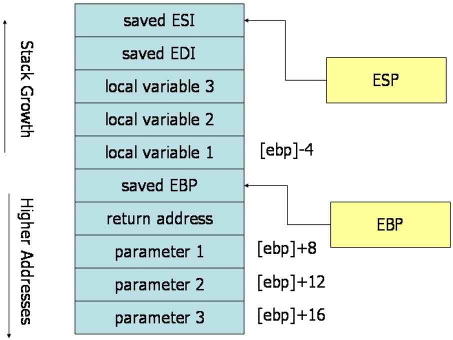


1. Push *caller-saved* registers: EAX, ECX, EDX
2. Push arguments in reverse order (allow varadics)
3. Use the `call` instruction (push return address, and jump)
4. Remove parameters from the stack (add their size to ESP)
5. Restore caller-saved registers (pop them from the stack)



Subroutine Prologue

1. Push EBP, and then copy ESP into EBP
   - All parameters are in EBP-offset
2. Allocate local variables in the stack
   - Subtract their size from ESP
   - All local variables are in EBP+offset
3. Push *callee-saved* registers: EBX, EDI, ESI



Subroutine Epilogue

1. Leave the return value in EAX
2. Restore callee-saved registers (pop them)
3. Deallocate local variables
   - Add their size to ESP
   - Better alternative, copy EBP into ESP
4. Restore the previous EBP (pop it)
5. Return to the caller by executing `ret`

Callee rules (3)

Instruction `leave` is equivalent to

- `mov esp, ebp`
- `pop ebp`

It is a shortcut for 3 and 4 in the previous slide.

Outline

Outline

Use assembly in C/C++

- Use instruction `__asm__(<assembly-code-here>)`
- Compile with `-masm=intel`

Example

Try `find_start.c`

Buffer overflow

- Essentially, writing after the last element of an array
- Target EIP to control execution of the running program

Example

1. Try `buffer.c`

Overflow into the return address

- Try `overflow.c`
- Function `gets()` does not bound its argument
- Find the address of function `unlinked_code`, say 0x0804845b
- Try the following:

```
for i in $(seq 30 50); do
   echo $i;
   python -c "print('A'*$i + '\x5b\x84\x04\x08')" | a.out;
done
```

Outline

Spawn a shell in C

- Try `shell.c`
- We modify the owner to root, and set the SUID bit

Spawn a shell in C

- Try `shell.c`
- We modify the owner to root, and set the SUID bit

What all this means

- Processes are associated with two user ids
  - Real UID: who started the process
  - Effective UID: for who the process acts
- Similarly, there are real and effective group ids

Spawn a shell in C

- Try `shell.c`
- We modify the owner to root, and set the SUID bit

What all this means

- Processes are associated with two user ids
  - Real UID: who started the process
  - Effective UID: for who the process acts
- Similarly, there are real and effective group ids
- If SUID is set, effective UID is set to the user owning the file
- If SGID is set, effective GID is set to the group owning the file
- `exec*` functions start new processes... acting for the effective user and group!

Our first shellcode

- A shellcode is a set of machine instructions
- Essentially, instructions spawning a shell
- Try `shellcode.c`

Our first shellcode

- A shellcode is a set of machine instructions
- Essentially, instructions spawning a shell
- Try `shellcode.c`

Problems we have to face

- Inject our shellcode in a vulnerable buffer
- Jump to the first instruction of our shellcode

Shellcode injection

- Try `victim.c`
- How to inject our shellcode?

Shellcode injection

- Try `victim.c`
- How to inject our shellcode?

The NOP Method

```
<NOPs (0x90)> <shellcode> <padding> <saved return address>
```

- We will jump in the NOP sled
- The more NOPs, the more likely the injection
- Follow the instructions

Outline

GDB enanchement

- There are a few enanchement for `gdb`
- They add pretty printing functionalities

GDB enanchement

- There are a few enanchement for `gdb`
- They add pretty printing functionalities
- One of them is `peda`
- Execute `source <path to peda.py>` in `gdb`

Download `peda` from github

`https://github.com/longld/peda`

Find offsets with `peda` (1)

- Use `pattern create` to create a long pattern

```
gdb-peda$ pattern create 1024
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAA
gAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAY
AAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A
%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%
WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsd
As3AsIAseAs4AsJAsfAs5AsKAsgAs6AsLAshAs7AsMAsiAs8AsNAsjAs9AsOAskAsPAslAsQAsmAsRAsoAsSAspAsTAsqA
sUAsrAsVAstAsWAsuAsXAsvAsYAswAsZAsxAsyAszAB%ABsABBAB$ABnABCAB-AB(ABDAB;AB)ABEABaAB0ABFABbAB1AB
GABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgAB6ABLABhAB7ABMABiAB8ABNABjAB9ABOABkABPABlABQABmABRABo
ABSABpABTABqABUABrABVABtABWABuABXABvABYABwABZABxAByABzA$%A$sA$BA$$A$nA$CA$-A$(A$DA$;A$)A$EA$aA
$0A$FA$bA$1A$GA$cA$2A$HA$dA$3A$IA$eA$4A$JA$fA$5A$KA$gA$6A$LA$hA$7A$MA$iA$8A$NA$jA$9A$OA$kA$PA$
lA$QA$mA$RA$oA$SA$pA$TA$qA$UA$rA$VA$tA$WA$uA$XA$vA$YA$wA$ZA$xA$yA$zAn%AnsAnBAn$AnnAnC'
gdb-peda$
```

Find offsets with `peda` (1)

- Use `pattern create` to create a long pattern

```
gdb-peda$ pattern create 1024
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAA
gAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAY
AAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A
%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%
WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsd
As3AsIAseAs4AsJAsfAs5AsKAsgAs6AsLAshAs7AsMAsiAs8AsNAsjAs9AsOAskAsPAslAsQAsmAsRAsoAsSAspAsTAsqA
sUAsrAsVAstAsWAsuAsXAsvAsYAswAsZAsxAsyAszAB%ABsABBAB$ABnABCAB-AB(ABDAB;AB)ABEABaAB0ABFABbAB1AB
GABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgAB6ABLABhAB7ABMABiAB8ABNABjAB9ABOABkABPABlABQABmABRABo
ABSABpABTABqABUABrABVABtABWABuABXABvABYABwABZABxAByABzA$%A$sA$BA$$A$nA$CA$-A$(A$DA$;A$)A$EA$aA
$0A$FA$bA$1A$GA$cA$2A$HA$dA$3A$IA$eA$4A$JA$fA$5A$KA$gA$6A$LA$hA$7A$MA$iA$8A$NA$jA$9A$OA$kA$PA$
lA$QA$mA$RA$oA$SA$pA$TA$qA$UA$rA$VA$tA$WA$uA$XA$vA$YA$wA$ZA$xA$yA$zAn%AnsAnBAn$AnnAnC'
gdb-peda$
```

- Crash the process using the pattern

```
Stopped reason: SIGSEGV
0x73413973 in ?? ()
gdb-peda$
```

Find offsets with `peda` (1)

- Use `pattern create` to create a long pattern

```
gdb-peda$ pattern create 1024
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAA
gAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAY
AAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A
%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%
WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsd
As3AsIAseAs4AsJAsfAs5AsKAsgAs6AsLAshAs7AsMAsiAs8AsNAsjAs9AsOAskAsPAslAsQAsmAsRAsoAsSAspAsTAsqA
sUAsrAsVAstAsWAsuAsXAsvAsYAswAsZAsxAsyAszAB%ABsABBAB$ABnABCAB-AB(ABDAB;AB)ABEABaAB0ABFABbAB1AB
GABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgAB6ABLABhAB7ABMABiAB8ABNABjAB9ABOABkABPABlABQABmABRABo
ABSABpABTABqABUABrABVABtABWABuABXABvABYABwABZABxAByABzA$%A$sA$BA$$A$nA$CA$-A$(A$DA$;A$)A$EA$aA
$0A$FA$bA$1A$GA$cA$2A$HA$dA$3A$IA$eA$4A$JA$fA$5A$KA$gA$6A$LA$hA$7A$MA$iA$8A$NA$jA$9A$OA$kA$PA$
lA$QA$mA$RA$oA$SA$pA$TA$qA$UA$rA$VA$tA$WA$uA$XA$vA$YA$wA$ZA$xA$yA$zAn%AnsAnBAn$AnnAnC'
gdb-peda$
```

- Crash the process using the pattern

```
Stopped reason: SIGSEGV
0x73413973 in ?? ()
gdb-peda$
```

- Use `pattern offset` to compute the offset

```
gdb-peda$ pattern offset 0x73413973
1933654387 found at offset: 524
```

Find offsets with `peda` (2)

- Use `pattern search` to find the address of the pattern

```
gdb-peda$ pattern search
Registers contain pattern buffer:
EIP+0 found at offset: 524
EBP+0 found at offset: 520
Registers point to pattern buffer:
[EDX] --> offset 1018 - size ~6
[ESP] --> offset 528 - size ~203
[ECX] --> offset 1018 - size ~6
Pattern buffer found at:
0xffffcc10 : offset    0 - size 1024 ($sp + -0x210 [-132 dwords])
0xffffd0b6 : offset    0 - size 1024 ($sp + 0x296 [165 dwords])
References to pattern buffer found at:
0xffffcbf0 : 0xffffcc10 ($sp + -0x230 [-140 dwords])
0xffffcc00 : 0xffffcc10 ($sp + -0x220 [-136 dwords])
0xffffcc04 : 0xffffd0b6 ($sp + -0x21c [-135 dwords])
gdb-peda$
```

Build skeletons for your exploits

- Use strings of the same length
- Script your exploit as much as possible
- Follow the instructions in `peda.txt`

Outline

Return to libc (ret2libc)

- Alternative to code injection
- Just inject return addresses (and arguments)

Return to libc (ret2libc)

- Alternative to code injection
- Just inject return addresses (and arguments)

Example

- Replace the return address with the address of `system()`
- Leave 4 bytes (it is the return address of `system()`)
- Write the address of the string to execute
- Follow the instructions in `ret2libc.txt`

Return to libc (ret2libc)

- Alternative to code injection
- Just inject return addresses (and arguments)

Example

- Replace the return address with the address of `system()`
- Leave 4 bytes (it is the return address of `system()`)
- Write the address of the string to execute
- Follow the instructions in `ret2libc.txt`

Return Oriented Programming (ROP)

Chain several calls to small instruction sets terminated by `ret`

Outline

Protection mechanisms

- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection

Protection mechanisms

- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection
- Canaries: memory after buffers store special values
  - Protect against buffer overflows
  - Usually randomized, and difficult to predict

Protection mechanisms

- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection
- Canaries: memory after buffers store special values
  - Protect against buffer overflows
  - Usually randomized, and difficult to predict
- AAAS: ASCII Armored Address Space
  - Start addresses of subroutines with \x00
  - Limit calls in case of overflows

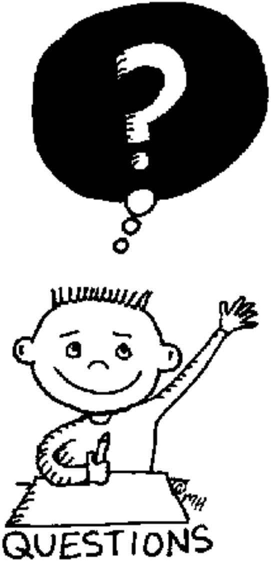
Protection mechanisms

- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection
- Canaries: memory after buffers store special values
  - Protect against buffer overflows
  - Usually randomized, and difficult to predict
- AAAS: ASCII Armored Address Space
  - Start addresses of subroutines with \x00
  - Limit calls in case of overflows
- ASLR: Address Space Layout Randomization
  - Randomly change addresses at each execution

Take-home message

- Vulnerabilities are due to security bug
- Protection mechanisms are introduced to stop common exploit on vulnerabilities
- New exploitation techniques are developed on top of previous techniques

If you don't update your programs,
you are exposed to several known vulnerabilities



END OF THE LECTURE