Buffer Overflow Attack (AskCypert CLaaS)

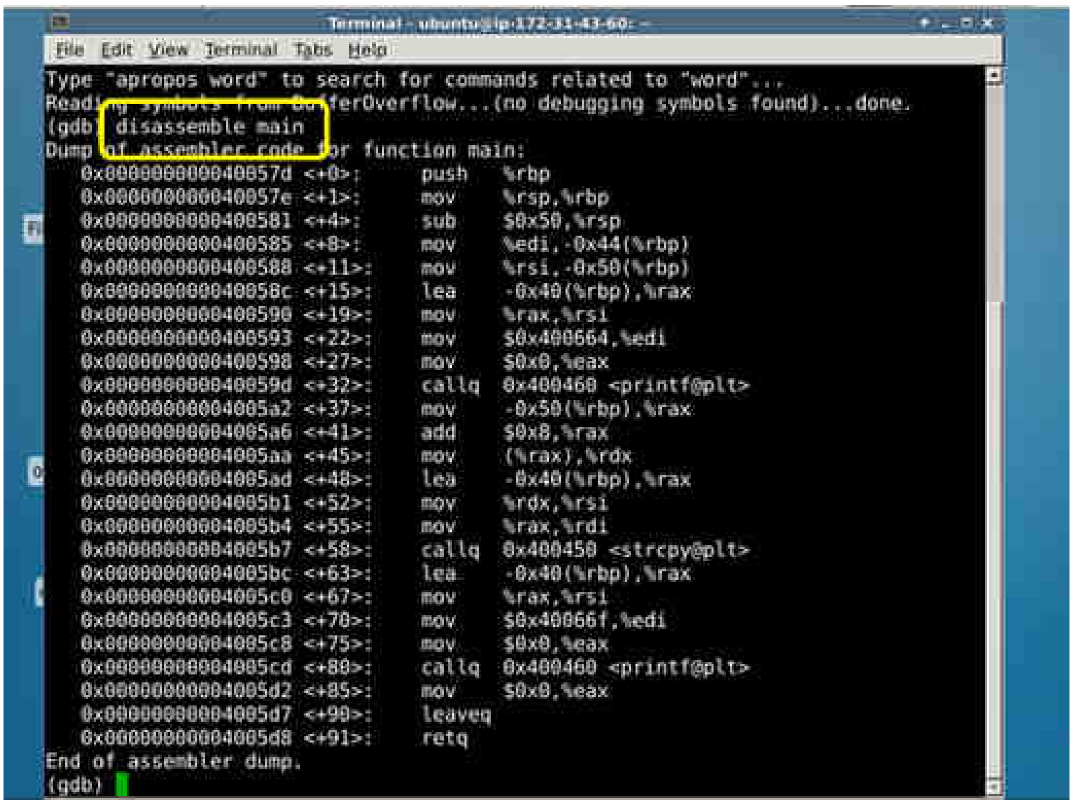
BufferOverflow.c code

```
1.  int main(int arg c, char** argv)
2.  {
3.  char name[64];
4.  printf("Addr;%p\n", name);
5.  strcpy(name, argv[1]);
6.  printf("Arg[1] is:%s\n", name);
7.  return 0
8.  }
```

Comments for dummies ☺ :

- Line 3: creation of a variable, called name, of type array of char, size = 64 bytes
- Line 4: print the memory address where name is stored
- Line 5: copy the string given as parameter of the function into the variable name
- Line 6: print the content of the variable name

Assembly code


Terminal - ubuntu@ip-172-31-43-60: ~
File Edit View Terminal Tabs Help

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from BufferOverflow...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040057d <+0>:     push   %rbp
   0x000000000040057e <+1>:     mov    %rsp,%rbp
   0x0000000000400581 <+4>:     sub    $0x50,%rsp
   0x0000000000400585 <+8>:     mov    %edi,-0x44(%rbp)
   0x0000000000400588 <+11>:    mov    %rsi,-0x50(%rbp)
   0x000000000040058c <+15>:    lea    -0x40(%rbp),%rax
   0x0000000000400590 <+19>:    mov    %rax,%rsi
   0x0000000000400593 <+22>:    mov    $0x400664,%edi
   0x0000000000400598 <+27>:    mov    $0x0,%eax
   0x000000000040059d <+32>:    callq  0x400460 <printf@plt>
   0x00000000004005a2 <+37>:    mov    -0x50(%rbp),%rax
   0x00000000004005a6 <+41>:    add    $0x8,%rax
   0x00000000004005aa <+45>:    mov    (%rax),%rdx
   0x00000000004005ad <+48>:    lea    -0x40(%rbp),%rax
   0x00000000004005b1 <+52>:    mov    %rdx,%rsi
   0x00000000004005b4 <+55>:    mov    %rax,%rdi
   0x00000000004005b7 <+58>:    callq  0x400450 <strcpy@plt>
   0x00000000004005bc <+63>:    lea    -0x40(%rbp),%rax
   0x00000000004005c0 <+67>:    mov    %rax,%rsi
   0x00000000004005c3 <+70>:    mov    $0x40066f,%edi
   0x00000000004005c8 <+75>:    mov    $0x0,%eax
   0x00000000004005cd <+80>:    callq  0x400460 <printf@plt>
   0x00000000004005d2 <+85>:    mov    $0x0,%eax
   0x00000000004005d7 <+90>:    leaveq
   0x00000000004005d8 <+91>:    retq
End of assembler dump.
(gdb)
```

Some notes and recalls about x64 assembly language

- When the stack grows (e.g., when a call to a function is done), addresses get smaller (cf. instruction <+4> in the listing above : sub $0x50, %rsp). That is, the bottom of the stack has address = 2^64-1 ; the top of the stack has address = 0

- Syntax convention is just a nightmare as there are many different conventions. There are 2 two main conventions: Intel convention, the original syntax for x86 processors which is dominant in the Windows world; and AT&T convention, dominant in the UNIX world. Examples of differences (useful to understand the BufferOverflow code) :

    - Register referencing: %rsp (ATT) vs rsp (Intel)
    - Immediate values: $13 (ATT) vs 13 (Intel)
    - Parameter order : source first, destination second (ATT) vs destination first, source second (Intel): mov $13, %rax (ATT) vs mov %rax, 13 (Intel) (store 13 in register rax)
- Registers (AT&T convention)
    - Function calling conventions
        - RDI, RSI, RDX, RCX, R8, R9, XMM0-7 (XMM: see below): by convention (System V-AMD64 convention (Linux, Unix…)), these registers are used to store the 6 first arguments (integer and pointer only for Rxx) of the called routine. 1st argument is stored in RDI;…; 6th argument is stored in R9
        - Additional arguments are pushed onto the stack. This must be done in reverse order (i.e., the last parameter first => the first parameter is stored at the lowest address)
        - Parameters less than 64 bits long are not zero-extended; high bits are just not considered
    - Caller-saved/callee-saved registers: some registers (RAX, RCX, RDX, R8-11 (Microsoft x64 calling convention) - RAX, RCX, RDX, RDI, RSI, RSP, R8-11 (AT&T convention) are said "caller-saved" (aka volatile registers). Before calling a function, the caller should save the content of these registers. Indeed as the called function (callee) is allowed to modify these registers, if the caller relies on their values after the function returns, the caller must push the values of these registers onto the stack, so that they can be restored. Other registers ("callee-saved registers) (RBX, RBP, RDI, RSI, RSP, R12-15 (Microsoft x64 calling convention) - RBX, RBP, R12-15 (AT&T x64 calling convention)) are considered as non-volatile and need not to be saved by the caller. It is the caller's responsibility to clean the stack after the call. That is why in the beginning of the code, BufferOverflow saves RDI and RSI (so as to be able to restore them after the further calls to printf and strcpy (RDI and RSI are caller-saved registers)) and why it saves RBP (as a callee-saved register, it is the duty of the function to save it before executing further instructions)
    - By replacing R by E in the name of the register (RDX $\rightarrow$ EAX; RIP $\rightarrow$ EIP; etc.), one can design the 32bit-lower part of RIP (useful for compliance with 32 bit-systems)

- The stack is aligned on 16 bytes boundaries (128 bits)

- Operating systems do not use the theoretical 2^64 (= 16 hexabytes) set of addresses. For instance, AMD64 processors support a physical address space of up to 2^48 bytes of RAM (i.e., only the 6 first bytes (48 bits) of the addresses are used)

- In x64, apart from general-purpose registers (see below), some registers are 128-bit long (XMM registers) (not used in our use case)

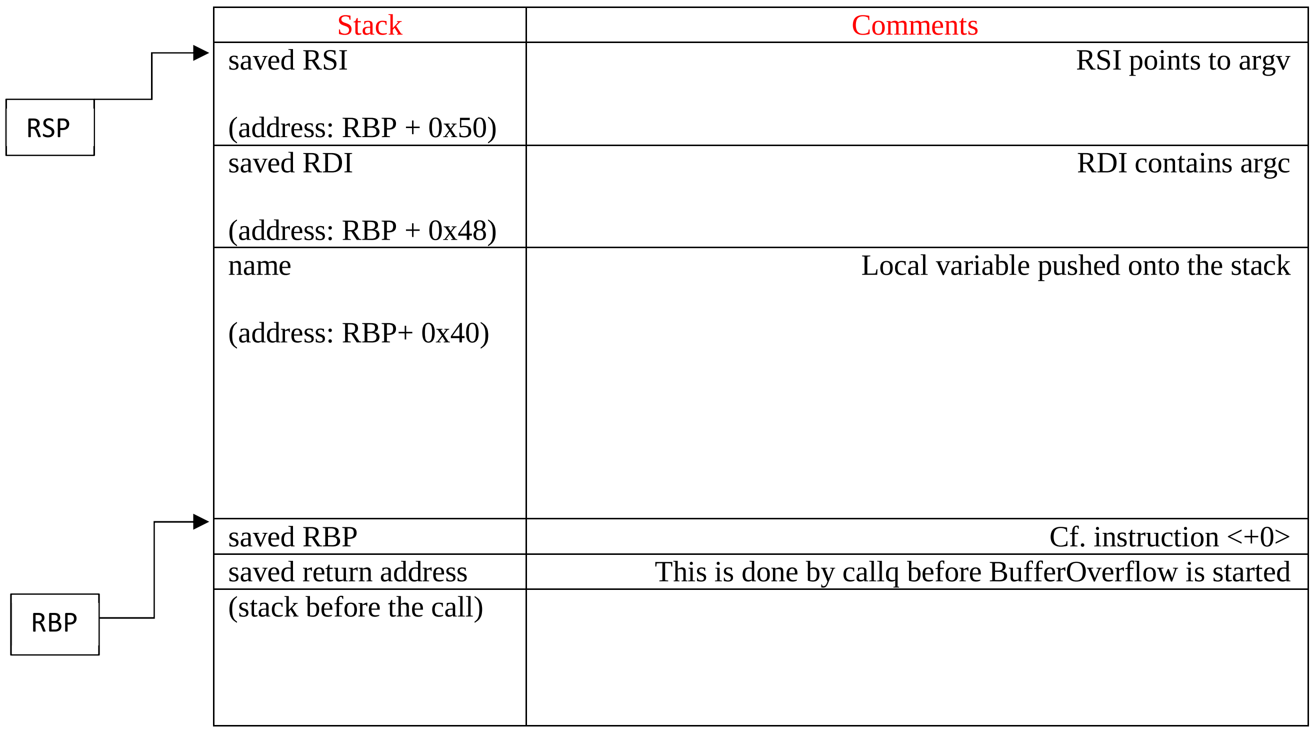
- General purpose 64 bit-registers – usage :
  - RSP : stack pointer
  - RIP : instruction pointer; contains the address of the next instruction
  - RBP : base pointer (aka frame pointer); points to the bottom of the stack frame allocated to the routine; the stack frame stores the local variables and the parameters passed to other function (call to an external function)
  - RAX : used to store the result of the function (if it exists and if it is less than 64 bit-long)
  - RBX : optionally used as a base pointer
  - EDI (32-bit lower part of RDI): used by gcc to store argc
  - RSI: used by gcc to store argv

- Before starting the execution of a subroutine, the callq instruction pushes the return address onto the stack (i.e., the actual value of the RIP pointer) , then it jumps to the address of the beginning of the subroutine

- The leaveq instruction, before the end of a subroutine, sets RSP to RBP, thus releasing the space allocated to the subroutine and restores RBP by popping it from the stack where it was saved at the beginning of the subroutine

- The retq instruction, at the very end of a subroutine, pops the return address from the stack into RIP, thus resuming the execution at the saved return address

Structure of the stack when BufferOverflow is running:



Stack	Comments
saved RSI (address: RBP + 0x50)	RSI points to argv
saved RDI (address: RBP + 0x48)	RDI contains argc
name (address: RBP+ 0x40)	Local variable pushed onto the stack
saved RBP	Cf. instruction <+0>
saved return address	This is done by callq before BufferOverflow is started
(stack before the call)	

Analysis of the assembly code of the BufferOverflow function

Prologue

- <0> : push %rbp : save the base pointer (callee-saved register) onto the stack

- <+1> : mov %rsp, %rbp : copy the value of the stack pointer into the base pointer; RBP points to the bottom of the frame stack

- <+4> : sub 0x50, %rsp : reserve space (80 bytes (0x50 = 80 decimal)) to save argc (stored in RDI) and argv (stored in RSI), and to store the local variable name. Name is 64-byte long, i.e., in hexa 40; RDI is 8 byte-long (0x8), and RSI is 8-byte long (0x8) => total size = 0x40+0x8+0x8 = 0x50 (or in decimal: 64+8+8 = 80 = 0x50)

- <+8> : mov %edi, -0x44(%rbp) : save EDI (the 32 bit lower part of RDI) at address -0x44(%rbp) (i.e., right on top of name et below RSI; see figure); as EDI contains argc, now argc is saved at address -0x44(%rbp)

- <+11> : mov %rsi, -0x50(%rbp) : save RSI at address -0x50(%rbp), i.e., on top of the frame stack (see figure); as RSI points to argv, now the content of address -0x50(%rbp) points to argv

Execution of printf(“Addr:%p\n:%s\n”, name)

- <+15> : lea -0x40(%rbp), %rax : copy the address of the variable name (cf. figure) into RAX

- <+19> : mov %rax, %rsi : copy RAX into RSI; now RSI points to name (recall: RSI is used to pass the 2nd argument to a subroutine, here to pass name to printf)

- <+22> : mov $0x400664, %edi : copy the address of the memory where the string “Addr;%p\n” is stored into EDI (recall: RDI is used to pass the 1st argument to a subroutine, here to pass the string “Addr;%p\n” to printf)

- <+27> : move $0x0, %eax : EAX stores the number of vector registers used to store arguments of the called function, here 0

- <+32> : callq 0x400460 <printf@plt> : call printf function (jump to the address of printf) (plt = procedure linkage table (= dynamic loading and linking) (0x400460 = address of printf)

Execution of strcpy(name, argv[1])

- <+37> : mov -0x50(%rbp), %rax : copy argv into RAX (see figure and instruction <+11>); now RAX points to argv[0]

- <+41> : add $0x8, %rax : by adding 0x8 to RAX, now RAX points to argv[1]

- <+45> : mov (%rax), %rdx : copy the value stored in rax i.e., the address of argv[1], into rdx

- <+48> : lea -0x40(%rbp), %rax : copy the address of the variable name (see figure and instruction <+11>) into RAX

- <+52> : mov %rdx, %rsi : copy RDX, which points to argv[1] (cf. instruction <+45>) into RSI (as before, RSI is used here to pass the 2nd argument to a subroutine; here to pass argv[1] to strcpy)

- <+55> : mov %rax, %rdi : copy RAX (which points to name; cf. instruction <+48>) into RDI (as before, RDI is used here to pass the 1st argument to a subroutine; here to pass name to strcpy)

- <+58> : callq 0x400450 <strcpy@plt> : call strcpy function ; strcpy copies its 2nd argument (actually the string pointed by its second argument) (argv[1] pointed by RSI) into its 1st argument (name, pointed by RDI)

Execution of printf(“Arg[1] is:%s\n”, name)

- <+63> : lea -0x40(%rbp), %rax : copy the address of the variable name (see figure and instruction <+11>) into RAX

- <+67> : mov %rax, %rsi : copy RAX, which points now to name (cf. instruction <+63>) into RSI (as before, RSI is used here to pass the 2nd argument to a subroutine; here to pass name to printf)

- <+70> : mov $0x40066f, %edi : copy the address of the memory where the string "Arg[1] is:%s\n" is stored into EDI (as before, RDI is used to pass the 1st argument to a subroutine, here to pass the string "Arg[1] is:%s\n" to printf)

- <+75> : mov $0x0, %eax : as in instruction <+27>, EAX stores the number of vector registers used to store arguments of the called function, here 0

- <+80> : callq 0x400460 <printf@plt> : call printf function (0x400460 = address of printf)

Output of BufferOverflow

- <+85> : mov $0x0, %eax : store the value 0 in EAX ; EAX stores the output of the called function, here 0 as the function was correctly executed

Epilogue

- <+90> : leaveq : release the space allocated to the subroutine and restore RBP by popping it from the stack (cf. instruction <+0>)
- <+91> : retq : pops the return address from the stack into RIP, thus resuming the execution at the saved return address

Analysis of the attack

As one can see on the figure, below the variable name (64 bytes), the stack stores RBP (8 bytes); below lies RIP i.e., the return address. If one passes a parameter of length larger than 64 bytes, then first RBP is overwritten; if the length of the parameter is larger than 72, then it is the return pointer RIP which is overwritten.

The goal of a buffer overflow attack is to overwrite the return pointer by an address which points to a shell/malicious code. This is done by overflowing a buffer, in our case, the name local variable.

To implement such an attack, one has to answer 2 questions:

- What is the "distance", on the stack, between the vulnerable buffer and the return pointer (note: the vulnerable buffer can be separated from the return pointer by other local variables/buffers)?
- What is the absolute address of the malicious code?

To answer the first question, two approaches are possible:

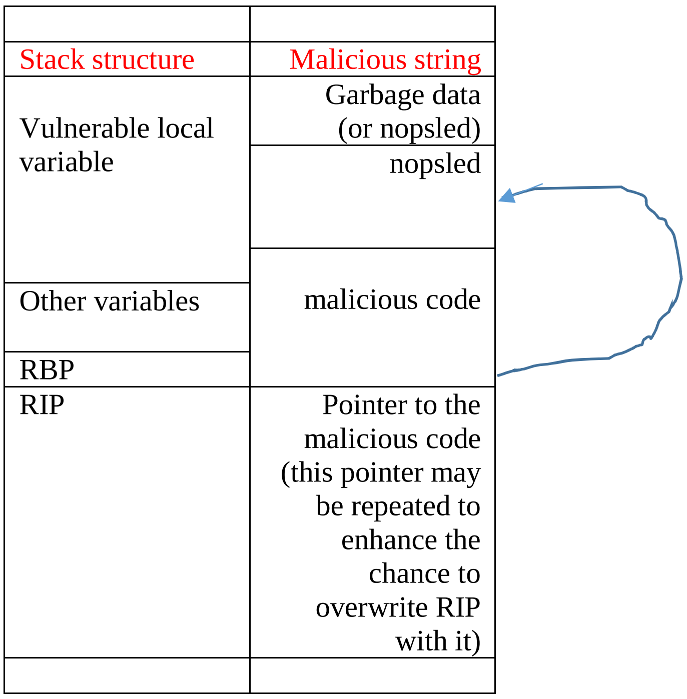
- Disassembling the code and reverse engineering the code, so as to determine the layout of the local variables that compose the frame stack
- Store into the vulnerable buffer a long pattern of characters, so that when the return pointer is overwritten and the function crashes, then one can determine (using, for instance, gdb)

the characters that were written into the return pointer; then, as the pattern of characters stored in the vulnerable buffer is known, it is trivial to compute the distance (in number of characters) between the vulnerable buffer and the return pointer

- If the attacker is not completely sure of its computation of this distance, a trick is to store, at the very end of the vulnerable buffer, a series of copies of the address that points to its malicious code (see figure below)

Answering the second question is not a trivial task as the attacker does not know the execution context i.e., the actual state of the stack.

A basic technique is then to use a “nopsled” i.e., a set of NOP (no operation) instructions:



Stack structure	Malicious string
Vulnerable local variable	Garbage data (or nopsled)
	nopsled
Other variables	malicious code
RBP	
RIP	Pointer to the malicious code (this pointer may be repeated to enhance the chance to overwrite RIP with it)

Note: if the address pointing to the malicious code is not correct but if it points to any address within the nopsled, the execution will "slide" down the NOP instructions until it reaches the malicious code which will then be executed.

Notes:

- The site metasploit can be used to get (classical) exploit codes
- Addresses should be written in little-endian (last characters first)
- The command x /nb $rsp allows one to visualize n bytes below RSP; it is useful to display the content of the stack