# Format string safety

## Summary

In this exercise, we will learn about format string safety.

## Prerequisites

Setup an Ubuntu VM as outlined in the VM setup instructions on Blackboard.

## Details

### Part I-Variadic function overview

Variadic functions (e.g., fprintf, printf, sprintf) receive a variable number of arguments and C/C++ does not validate the number of arguments passed in at runtime. Formatted string functions are variadic function that interpret variable argument lists according to an input string that is also an argument and it specifies the variable argument list. Format strings are character sequences consisting of ordinary characters (excluding %) and conversion specifications (e.g., %s, %d, %x, %n).

Allowing the format string input to be user provided is very dangerous. Unfortunately, many programmers are unaware of the malicious capabilities enabled by allowing the above. User supplied format strings that write to a character array (e.g., sprintf) could be susceptible to buffer overflow attacks, reading arbitrary data from the stack and memory, and even writing arbitrary data to memory.

### Part II-Leaking data from the stack (dumb method)

The dumb information leak pops values off the stack by adding false arguments (via conversion specifiers) in the user-controlled format specifier passed to printf.

#### Demo

* Download the project into a local sandbox

$ git clone <https://gitlab.com/underpantsgnomes/softwaresecurity/formatstringexploit.git>

* Turn off address-space layout randomization in Ubuntu (ASLR) using

$ echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

* Ensure ASLR is disabled by confirming the output of running the following command is 0

$ cat /proc/sys/kernel/randomize\_va\_space

* Import the project in Eclipse
  + File->Open Projects From File System
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C compiler
  + Compile separately prior to a later linking step (-c)
  + Enable position independent code (-fPIC)
  + Specify to retain frame pointers (-fno-omit-frame-pointer); note: targeted leaf functions (ones that don’t call anything else) might not retain frame pointers even with this setting
  + Disable format string compile time warnings (-Wno-format-security); note: prevents build errors when -Werror is specified
  + Disable Ubuntu’s GCC-SSP enabling of run-time stack overflow verification via stack canaries (-fno-stack-protector)
  + Specify intel dialect for outputted assembly (-masm=intel)
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C linker
  + Enable executable stack for the linker (-z execstack)
* exploitWrapper.c has four alternative modes, set the leaking information off the stack (dumb version) by enabling its preprocessor macro and disabling the other three modes. vulnerableUUT.c has three alternative modes, set it to call vulnFunction1 and disable the calls to the other two vulnerable functions.
* Inspect the exploit code
* Build the project
* Create the debug target configuration
* Select new launch configuration for a C/C++ Application
* Specify the wrapper binary built from the console as the target
* Insert the following three program arguments

|  |
| --- |
| ${string\_prompt: shellcodeaddr:0xdeadbeef} ${string\_prompt: oldrbp:0xdeadbeef} |

* Go to the Debugger tab and specify an environment that uses ~/.gdbinit
* Set a breakpoint on line 31 in vulnFunction1::vulnerableUUT.c
* Run the demo
* Continue to the breakpoint and locate rsp for vulnFunction1::vulnerableUUT.c
* Set assembly instruction stepping mode and switch over to disassembly view
* Step into \_\_printf and then step into \_IO\_vfprintf\_internal
  + Locate rsp in \_IO\_vfprintf\_internal
* Step return out of \_IO\_vfprintf\_internal and then the same for \_\_printf
* Look at the values leaked (printed) to the console and locate these in stack memory

### Part III-Leaking arbitrary data (smart method)

The smart information leak uses a %s conversion specifier within a tainted format specifier to read a string from a user controlled address. As 64-bit for the oftentimes have 0x00 bytes, we have relaxed assumptions for this example and assume that the read address can be placed on the stack separate from the tainted format specifier. With the desired read address already on the stack, the tainted format specifier pops values off the stack to align rsp with the read address for the %s conversion specifier used by printf. False arguments (via conversion specifiers) are added in the user-controlled format specifier to align rsp and to obtain the string at the specified read address.

#### Demo

* In exploitWrapper.c, set the arbitrary read (smart version) by enabling its preprocessor macro and disabling the other three modes. vulnerableUUT.c has three alternative modes, set it to call vulnFunction1 and disable the calls to the other two vulnerable functions.
* Inspect the exploit code
* Build the project
* Run the demo
* Continue to the breakpoint and locate rsp for vulnFunction1::vulnerableUUT.c
* Set assembly instruction stepping mode and switch over to disassembly view
* Step into \_\_printf and then step into \_IO\_vfprintf\_internal
  + Locate rsp in \_IO\_vfprintf\_internal
* Step return out of \_IO\_vfprintf\_internal and then the same for \_\_printf
* You should receive a segmentation fault for passing in a read address of 0xDEADBEEF
* Terminate and relaunch the program and view memory to locate your environment variables and identify an address for an interesting string to leak
* Terminate and relaunch the program and this time pass in the desired read address to the oldrbp prompt input to exploitWrapper
* Look at the data leaked (printed) to the console

### Part IV-Smashing the stack with a sledge hammer

The stack smashing sledge hammer is similar to our previous stack smashing exploit except this time it is implemented using a malicious format specifier. False arguments, via conversion specifiers that specify extremely excessive digit padding, are added in the user-controlled format specifier to overflow locals.outbuffer and corrupt the stack frame return fptr in the second call to sprintf.

#### Demo

* In exploitWrapper.c, set sledge hammer stack smashing mode by enabling its preprocessor macro and disabling the other three modes. vulnerableUUT.c has three alternative modes, set it to call vulnFunction2and disable the calls to the other two vulnerable functions.
  + Inspect the exploit code
* Set a breakpoint on line 44 in vulnFunction2::vulnerableUUT.c
* Run the demo
* Continue to the breakpoint and locate return fptr in the stackframe for vulnFunction2::vulnerableUUT.c in a memory window
* Step over the sprintf call and watch the stack get smashed

### Part IV-Corrupting the stack with tweezers

The tweezer approach uses 8 selectively placed %n conversion specifiers within a tainted format specifier to write 8 contiguous bytes starting at a user specified base address. As 64-bit addresses oftentimes have 0x00 bytes, we have again relaxed assumptions for this example and assume that the write addresses used by the %n conversion specifiers can be placed on the stack separate from the tainted format specifier. With the desired write addresses already on the stack, the tainted format specifier pops values off the stack to align rsp with the first address for the first %n conversion specifier used by sprintf . False arguments (via conversion specifiers) are added in the user-controlled format specifier to align rsp and to write the individual bytes over the stack frame’s return pointer so that rip is eventually redirected to our shellcode. There are special padding characters inserted between the %n conversion specifiers that result in appropriate byte counts modulo 256 that match our bytes in the shellcode address being written to the return fptr. The general organization of the tainted format specifier is shown below.

Table -Layout of snprintf's stack memory (after vulnFunction calls it) prior to processing the tainted format specifier

|  |
| --- |
| shellcode+spacer+padding1+write1cmd+padding2+write2cmd+padding3+write3cmd+padding4+write4cmd+padding5+write5cmd+padding6+write6cmd+padding7+write7cmd |

#### Demo

* To control the rip via this, we want to overflow and overwrite the return pointer with our shellcode address. Even if stack protections are disabled, when the epilogue runs, the shellcode will be executed.
* Inspect the exploit code and discuss amongst your team
* Run the program in Eclipse and step through the assembly instructions of the shellcode when it is called
  + If it segfaults, you need to adjust shellcodeaddr and oldrbp passed in to exploitwrapper appropriately (also ensure that ASLR is disabled in OS and also in gdb via ~/.gdbinit)
  + Execute the following in the bash shell when your shell code call works (i.e., when you get the $ bash prompt in your debugging session)

|  |
| --- |
| $ /usr/games/cowsay -f dragon "Grrr!"  < Grrr! >  --------  \ / \ //\  \ |\\_\_\_/| / \// \\  /0 0 \\_\_ / // | \ \  / / \/\_/ // | \ \  @\_^\_@'/ \/\_ // | \ \  //\_^\_/ \/\_ // | \ \  ( //) | \/// | \ \  ( / /) \_|\_ / ) // | \ \_\  ( // /) '/,\_ \_ \_/ ( ; -. | \_ \_\.-~ .-~~~^-.  (( / / )) ,-{ \_ `-.|.-~-. .~ `.  (( // / )) '/\ / ~-. \_ .-~ .-~^-. \  (( /// )) `. { } / \ \  (( / )) .----~-.\ \-' .~ \ `. \^-.  ///.----..> \ \_ -~ `. ^-` ^-\_  ///-.\_ \_ \_ \_ \_ \_ \_}^ - - - - ~ ~-- ,.-~  /.-~ |

* Rebuild the project and enable format string warning protections and set -Werror. What happens when you run try to build?
* Revert back to original build settings and add -D\_FORTIFY\_SOURCE=2 and set optimizations to -O1. Rebuild TWEEZERS and run it. What happens?

## References

* 1989, Miller et al., An Empirical Study of the Reliability of UNIX Utilities
* [1999, Tymm Twillman, Exploit for proftpd 1.2.0pre6](https://seclists.org/bugtraq/1999/Sep/328)
* [2000, tf8(), WuFTPD: Providing \*remote\* root since at least1994](https://seclists.org/bugtraq/2000/Jun/297)
* [2000, Tim Newsham, Format String Attacks](https://web.archive.org/web/20010802165341/http:/www.guardent.com/docs/FormatString.PDF)
* [2001, Scut, Team Teso, Exploiting Format String Vulnerabilities](https://web.archive.org/web/20011215135054/http:/www.team-teso.net/articles/formatstring/)
* 2001, Cowan et al., FormatGuard: Automatic Protection From printf Format String Vulnerabilities
* [2014, System V Application Binary Interface, AMD64 Architecture Processor Supplement](https://www.uclibc.org/docs/psABI-x86_64.pdf)
* 2013, Seacord, Secure Coding in C and C++, 2e, 2.3-2.4 (Buffer overflows from string vulnerabilities)
* <https://wiki.ubuntu.com/Security/Features>