Lecture 7

Strings. Information Leaks

Computer and Network Security
November 11, 2019

Computer Science and Engineering Department

Stack Frame

- created for each function call
- caller stores function arguments in registers or on stack
- issues `call` $\rightarrow$ saves instruction pointer and jumps to function code
- calee saves frame pointer, points frame pointer to current stack top and decrements stack pointer (increses the stack)
- the other way around for returning from a function call

Stack Pointer

- stack top
- stack pointer decreases → stack grows
- stack pointer increases → stack shrinks
- `esp` on x86
- `rsp` on x86_64
- `push` and `pop` instructions

- instruction to run
- value at a given time is the address of the next instruction (next to the one being currently run)
- affected by `jmp` & friends, `call` and `ret`
- needs to point to an executable memory area
- may point to an injected code to trigger an exploit

- contiguous memory area; array of bytes
- possesses: base address, length, type
- operations: allocate, free, index, get, set, copy to/from
- exploitable through: bounds overflow (buffer overflow) and wrong index (index out of bounds)
- exploits often make use of string buffers

- set of machine code instructions running as an exploit
- injected by the attacker in the stack, heap or another area
- the area needs to be executable
- instruction pointer is set at the beginning of the shellcode
- usually it runs an `execve("/bin/bash", "/bin/bash")` call

- static & dynamic analysis
- ASCII armored address space
- stack guard, canary value
- DEP: Data Execution Prevention
- ASLR: Address Space Layout Randomization

- memory address
- array
- array of characters
- ends with null character (`'\0'`)
- data exchange between program and user/environment
- difference between code and data at “primary level” – non-existent

What Is a Character?

- a singular element of a string
- not inherently signed or unsigned
- character data – used for strings
- representation (number of bits/bytes) may depend on hardware architecture and compiler

- byte character types: char, signed char, unsigned char
- char may be defined as either signed char or unsigned char
- char is distinct
- char is the type of each element of a literal
- char is used for character data

- what kind of data type is `EOF`?
- what kind of data type is `'a'` or `'\0'`?
- what happens when you compare chars with int?
- why does `fgetc` return an int? why does `isalpha()` receive an int as argument?
- always cast char to unsigned char for string comparisons

Null-Terminated Byte Strings (NTBS)

- naming from Robert Seacord (Secure Coding Initiative at CERT)
- use *null character* or `NUL byte` (`'\0'`) for ending strings
- length is number of characters, excluding *null character*
- string has to fit into a memory/buffer/array, otherwise it exceeds bounds

- `char` is used for strings
- only `=`, `==`, `!=` should be used for `char`
- comparisons must be handled by `signed char` or `unsigned char`

- allocation: static, dynamic
- initialization
- copying
- concatenating
- duplicating
- truncating
- browsing
- find length

- you always need to know string length
- a proper string needs to be null-terminated
- never go past a string
- buffer overflows and other kinds of attacks are due to exceeding string bounds

- a proper string ends in the `'\0'` character
- if missing null termination, string operations will go crazy
- any string operation ends at null termination

Truncation

- functions such as `strncpy` truncate the string
- string truncation may cause exploits – truncate the string at the right time and append something else
- if truncation occurs, the programmer must be aware of it and treat it accordingly
- string size must always be known

- due to bad computation, a value may be increased or decreased with a unit
- that may be the string length or placement of the null terminator

- some characters may be invalid for current processing
- see SQL injection attacks
- string should be validated
- white listing or black listing
- null terminators inside the string

- make sure you don’t break the initial string
- avoid strtok and strsep
- should be done by a Bison/Flex or a custom parser that is able to fully browse the whole string
- while tokenizing have in mind the other issues with strings

- needs to always be known
- any string operation functions are there to make it easy for the programmer, not to assume string length
- most string management functions may be replaced by `memcpy()`

- aim for an exploit
- run arbitrary code
- pass a condition
- execute shellcode

- stack
- stack frame
- buffer overflow
- return address
- shellcode

- go past string boundary
- when using `gets` (deprecated in C99, removed in C1X)
- when copying strings
- overwrite
  - variable value
  - function pointer data
  - return address

- write variable or function pointer through buffer overflow
- code injection, arbitrary code run, run code on stack/heap, shell code
- return-to-libc (arc injection) aim for `system()` or `exec()`

Input Validation

- input must not be trusted
- always check string content and string size
- be on the lookout for
  - invalid characters
  - strings that are too large
  - string truncation
- input is
  - command line arguments
  - environment variables
  - standard input
  - files, sockets and pipes

- caller allocates, caller frees – `strcpy`
- callee allocates, caller frees – `strdup`
- callee allocates, callee frees – init and destroy functions, constructors and destructor methods

Consistency

- make sure you use the memory management model for strings
- use the same functions in the same way
- check using the same approaches
- if required, define custom string management functions and use those

Better String Management Functions

- `strncpy` (ANSI), `strlcpy` (BSD), `strcpy_s` (Windows)
- these functions are not bullet proof
- `strncpy` solves out of bounds problems
- `strlcpy` is better than `strncpy`: solves missing `Null` termination
- string truncation is still a problem
- a programmer still needs to know string size
- these functions don't make a good programmer out of a bad programmer

- needs to always be known (yup, it's the third time we say this)
- know size of the **whole** string; beware of
  - `'\0'` characters in string
  - string truncation
- beware of `sizeof()` vs. `strlen()`
  - `sizeof(a) == strlen(a)`, if `a` is an array
  - `sizeof(a) != strlen(a)`, is `p` is a pointer

- We use Python for input generation since first lab
- `encode()`/`decode()` handles hex representation of characters
- lambda functions on string characters using `join()`
- list slicing using [x:y]
- list indices, also negatives

- `p32()` and `p64()` format addresses like its original representation in memory (endianness and sign)
- `unpack` function translates back to unpacked number depending on the data size, endianness and sign
- alternative: `pack` and `unpack` functions from `struct module`

- string formats are used to know how to show data and its size
- if the format can be manipulated by program input, private data can be read

Leaking with puts

- `puts` reads parameter data from stack until terminating null byte
- if the parameter string is not properly placed in memory, puts will read bytes and leak important information
- buffers are placed under old ebp and return address in process memory layout
- usually this data can be leaked using `puts`

- GOT stores library address
- GOT address is known for non-PIE executables
- usual to leak GOT `puts` address using `puts`

- `printf` called without format parameter can let us place our own format
- `printf(buf);` considering buf is read from input
- printf reads parameters from stack, by format

- printf format argument %x - prints a number in hex format
- printf format argument %n - writes the number of bytes written. The number is placed at the address given as parameter
- Enough to read and write memory if the attacker has access to format parameter

- STR00-C to STR10-C on “07. Characters and Strings” in CERT C Secure Coding Standard

- STR30-C to STR38-C on “07. Characters and Strings” in CERT C Secure Coding Standard

Keywords

- string
- character
- `char`, `signed char`, `unsigned char`
- NTBS
- null character
- character operators
- string operations
- bounds
- overflow
- truncation
- sanitization
- `gets`
- exploit
- input validation
- memory model

- CERT C Secure Coding Standard – 07. Characters and Strings (STR) – `https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=271`
- Secure Coding in C and C++ Class
  - Module 1. Strings
- Secure Coding in C and C++
  - Chapter 2. Strings