Assignment 5

Analysis of Network Installation

Mar 11th, 2024

Executive Summary

In my network installation task, I've employed a suite of diagnostic and security analysis tools to ensure the robustness and security of my system. Throughout this task, my primary objective was to establish a secure and optimized network infrastructure capable of resisting potential security threats.

I utilized the strings command to extract readable strings from binaries, which revealed aspects of the server.c program's functionality, such as networking and error handling. This tool is instrumental in identifying human-readable content that can sometimes expose sensitive information or hints about the program's functionality, which is critical for reinforcing security.

The nm tool provided a detailed map of the program's symbols, offering insight into the compiled binary's behaviour and dependencies. It helped me identify the structure and potential vulnerabilities without having to delve into the source code, ensuring that the network installation was not compromised by hidden risks.

With objdump, I delved into the program's assembly-level operations and memory layout, unveiling external dependencies and operational details. This perspective is essential for identifying inefficiencies and vulnerabilities that could be exploited in a network environment.

Strace and ltrace allowed me to trace system calls and library calls, respectively. By monitoring these calls, I gained visibility into the application's interaction with the operating system and its use of shared libraries. This information is vital for identifying security weaknesses and ensuring the application adheres to best practices in secure coding and system interaction.

Lastly, I employed VirusTotal to scan files and URLs for malware using various antivirus engines. The reports indicated that the scanned files were not detected as malicious by any of the engines. However, I remain aware that no detection is not a definitive all-clear, and I continue to exercise other security measures as part of a comprehensive defence strategy.

Chkrootkit served as my first line of defence against rootkits. Its detailed output informed me of the absence of known rootkits and flagged suspicious files and directories for further investigation, reinforcing the need for ongoing vigilance in network security.

These critical findings highlight the importance of thorough software analysis and proactive security measures in installing and maintaining network infrastructure. By leveraging these tools, I've underscored the implications of each discovery and fortified our network against a range of security threats.

Introduction

The scope of the network installation endeavour was defined to include setting up a robust and secure network infrastructure capable of supporting a variety of organizational needs while safeguarding sensitive data. The primary objective was to ensure that every component of the network, from the hardware to the application layer, was configured to offer optimal performance and resistance to cybersecurity threats. This phase is crucial to protecting the network from malicious activities and ensuring the confidentiality, integrity, and availability of data.

Understanding that a network's security is as strong as its weakest link, this installation phase was approached with a comprehensive strategy. It encompassed systems' physical setup and configuration, establishing protocols for safe data transmission, and using tools to detect and respond to potential security breaches. These proactive steps dictate a network's resilience against future security challenges.

Throughout the process, I was conscientious about adhering to ethical standards and cybersecurity best practices. I followed ethical guidelines to avoid unauthorized access or data manipulation. These principles were integral to maintaining and ensuring the network installation supports a secure and ethically operational environment. I did not use any malicious code in my operation; the code I used was a secure code from my COMP 7005 Assignment.

Body

Strings:

- Running the strings command on my server.c program highlighted crucial text-based data, detailing network functions and system library dependencies, essential for ensuring security and compatibility in the network environment.

Nm:

- By deploying nm in my program, I mapped out and categorized symbols, gaining insights into function behaviors and dependencies, which is instrumental in security analysis and debugging.

Objdump:

- Using objdump allowed me to dissect the binary structure and assess the assembly code, providing a deeper understanding of the application's performance and potential security vulnerabilities.

Strace:

- With Strace, I captured the system calls during the operation of my server application, providing a granular view of OS interactions that inform security and performance optimizations.

Ltrace:

- Ltrace gave me a window into the library calls within my server application, which is critical for identifying and rectifying potential security vulnerabilities linked to external libraries.

VirusTotal:

- I leveraged VirusTotal to scan my files, using multiple antivirus engines to ensure malware-free software, a fundamental step in the network security protocol.

Chkrootkit:

- Executing chkrootkit afforded me an overview of the system's integrity, identifying potential rootkit infections and reinforcing the first line of defence in my network installation process.

strings

```
sami@sami-mini-pc-ubuntu:~/Documents/socket1/cmake-build-debug$ strings server
/lib64/ld-linux-x86-64.so.2
inet_ntop
__cxa_finalize
read
malloc
__libc_start_main
strtoumax
fprintf
optopt
socket
inet_pton
fopen
getopt
fclose
memset
sigemptyset
puts
free
fflush
realloc
bind
opterr
htons
snprintf
stderr
sigaction
listen
optind
poll
strerror
perror
accept
__errno_location
putchar
exit
fwrite
__stack_chk_fail
setsockopt
libc.so.6
GLIBC_2.4
GLIBC_2.34
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
huVH
```

The strings command extracts printable strings from a binary or non-text file. It searches for sequences of characters that are at least four characters long (by default). It consists solely of printable characters, followed by an unprintable character (often a null byte, marking the end of a string in C). This tool is useful for identifying human-readable content in binary files, such as error messages, file paths, and other textual data that can provide context or clues about the file's functionality or data it might work with.

The strings command on my server.c program revealed information about networking functions (such as socket, bind, listen, accept), standard library functions (malloc, realloc, free, exit), error-

handling routines (perror, fprintf to stderr), and file operations (fopen, fclose, fwrite). This output clearly depicted my program's backbone in terms of network communication, memory management, error handling, and file manipulation capabilities.

The presence of strings linked to the GNU C Library (e.g., /lib64/ld-linux-x86-64.so.2, libc.so.6, and various GLIBC version strings) underscored the binary's reliance on standard Linux system libraries, emphasizing the need for a compatible execution environment. Networking functions highlighted within the output present the program's essence as a network server, equipped to establish sockets and manage client connections.
Memory management and error handling strings such as malloc, realloc, free, and perror illuminated dynamic memory allocation and runtime error management approaches. These are fundamental to ensuring the application's robustness and security, under heavy network traffic or malicious attempts to exploit memory management vulnerabilities.
Additionally, user interaction and error message strings (Unknown option '-%c', Missing arguments., etc.) depicted the server's user interface and mechanism for guiding users and reporting errors—elements that enhance usability and security by providing clear feedback and instructions.
Strings related to file operations showed the server's functionality concerning data receipt, processing, and storage, about the application's data handling strategies and potential areas for security enhancements.

The exploration with strings deepened my understanding of how various program components are manifested within the compiled binary. Each extracted string shed light on the program's capabilities and how it interacts with the system, manages user input, and navigates errors. This underscores the role of tools like strings in software analysis, debugging, and security assessment. Understanding the content within binaries is pivotal for uncovering hidden functionalities, dependencies, or even hardcoded sensitive information, thus informing the security measures necessary to safeguard server applications in a networked environment. My analysis confirms the significance of strings in understanding and securing the essence of networked applications.

Nm

```
sami@sami-mini-pc-ubuntu:~/Documents/socket1/cmake-build-debug$ nm server
000000000000038c r __abi_tag
                 U accept@GLIBC_2.2.5
                 U bind@GLIBC_2.2.5
0000000000005010 B __bss_start
                 U close@GLIBC_2.2.5
0000000000005048 b completed.0
0000000000001e64 t convert_address
                 w __cxa_finalize@GLIBC_2.2.5
0000000000005000 D __data_start
0000000000005000 W data_start
0000000000001490 t deregister_tm_clones
0000000000001500 t __do_global_dtors_aux
0000000000004cc0 d __do_global_dtors_aux_fini_array_entry
0000000000005008 D __dso_handle
0000000000004cc8 d _DYNAMIC
0000000000005010 D _edata
0000000000005050 B _end
                 U __errno_location@GLIBC_2.2.5
000000000000504c b exit_flag
                 U exit@GLIBC_2.2.5
                 U fclose@GLIBC_2.2.5
                 U fflush@GLIBC_2.2.5
00000000000028d8 T _fini
                 U fopen@GLIBC_2.2.5
                 U fprintf@GLIBC_2.2.5
0000000000001540 t frame_dummy
0000000000004cb8 d __frame_dummy_init_array_entry
0000000000003654 r __FRAME_END__
                 U free@GLIBC_2.2.5
                 U fwrite@GLIBC_2.2.5
                 U getopt@GLIBC_2.2.5
0000000000004eb8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000003344 r __GNU_EH_FRAME_HDR
0000000000001ba9 t handle_arguments
0000000000002848 t handle_disconnection
000000000000214e t handle_new_client
                 U htons@GLIBC_2.2.5
                 U inet_ntop@GLIBC_2.2.5
                 U inet_pton@GLIBC_2.2.5
0000000000001000 T _init
0000000000003000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
                 U listen@GLIBC_2.2.5
0000000000001549 T main
                 U malloc@GLIBC_2.2.5
                 U memset@GLIBC_2.2.5
0000000000005028 B opterr@GLIBC_2.2.5
0000000000005020 B optind@GLIBC_2.2.5
0000000000005024 B optopt@GLIBC_2.2.5
```

My utilization of nm involves running it against compiled programs to list the symbols from those files. This presents a map of function names, variable names, and other identifiers and categorizes them with symbol types and addresses. Symbol types ranging from "T" for text (code), "B" for uninitialized data (bss), "D" for initialized data, to "U" for undefined symbols offer a wealth of information. They help me understand the program's behaviour, dependencies, and architectural decisions without delving into the source code.

Undefined Symbols (U): Observing symbols like accept@GLIBC_2.2.5 and bind@GLIBC_2.2.5 as undefined within the object file was telling. It indicated dependencies on functionalities defined elsewhere, typically in shared libraries such as GLIBC. This is crucial for network operations, memory management, and file operations, illuminating the external dependencies that underpin the program's network communication capabilities.
Text Section (T, t): Symbols in this category represent the program's executable functions. Seeing symbols for main, convert_address, handle_arguments, and others provided insight into the program's operational aspects, from the entry point to network communication and file handling. The distinction between local (lowercase t) and global (uppercase T) symbols further helped understand these functions' scope and linkage.
Data Sections (D, d, B, b): Symbols related to data storage and variables, such as __bss_start, __data_start, and exit_flag, outline the variables that the program uses, distinguishing between those initialized and uninitialized. Thus, the symbols shed light on the application's memory footprint and management strategies.
Weak Symbols (W, w): Encountering weak symbols like _ITM_deregisterTMCloneTable hinted at the application's use of transactional memory or other advanced features. This flexibility in linkage and override capabilities offered insights into the complexity and modernity of the application's build.
Nm is indispensable for debugging, optimizing, and conducting a security assessment. Understanding the structure, dependencies, and behaviours embedded within the binary enables a more targeted approach to secure the application. It informs the identification of potential vulnerabilities linked to external dependencies or the implementation of specific functions.

My exploration with nm demystified my server program's structural and operational aspects and reinforced the interconnectedness of security, performance, and application architecture. This tool has become an integral part of my toolkit, essential for ensuring that my applications are functional and fortified against the threats in the networked world.

Objdump

```
sami@sami-mini-pc-ubuntu:~/Documents/socket1/cmake-build-debug$ objdump -d server

server:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 3f 00 00    mov    0x3fd9(%rip),%rax        # 4fe8 <__gmon_start__@Base>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 9a 3e 00 00       push   0x3e9a(%rip)        # 4ec0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       f2 ff 25 9b 3e 00 00    bnd jmp *0x3e9b(%rip)        # 4ec8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102d:       0f 1f 00                nopl   (%rax)
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   $0x0
    1039:       f2 e9 e1 ff ff ff       bnd jmp 1020 <_init+0x20>
    103f:       90                      nop
    1040:       f3 0f 1e fa             endbr64
    1044:       68 01 00 00 00          push   $0x1
    1049:       f2 e9 d1 ff ff ff       bnd jmp 1020 <_init+0x20>
    104f:       90                      nop
    1050:       f3 0f 1e fa             endbr64
    1054:       68 02 00 00 00          push   $0x2
    1059:       f2 e9 c1 ff ff ff       bnd jmp 1020 <_init+0x20>
    105f:       90                      nop
    1060:       f3 0f 1e fa             endbr64
    1064:       68 03 00 00 00          push   $0x3
    1069:       f2 e9 b1 ff ff ff       bnd jmp 1020 <_init+0x20>
    106f:       90                      nop
    1070:       f3 0f 1e fa             endbr64
    1074:       68 04 00 00 00          push   $0x4
    1079:       f2 e9 a1 ff ff ff       bnd jmp 1020 <_init+0x20>
    107f:       90                      nop
    1080:       f3 0f 1e fa             endbr64
    1084:       68 05 00 00 00          push   $0x5
    1089:       f2 e9 91 ff ff ff       bnd jmp 1020 <_init+0x20>
    108f:       90                      nop
    1090:       f3 0f 1e fa             endbr64
    1094:       68 06 00 00 00          push   $0x6
    1099:       f2 e9 81 ff ff ff       bnd jmp 1020 <_init+0x20>
    109f:       90                      nop
    10a0:       f3 0f 1e fa             endbr64
    10a4:       68 07 00 00 00          push   $0x7
    10a9:       f2 e9 71 ff ff ff       bnd jmp 1020 <_init+0x20>
```

objdump works by analyzing compiled binary files and presenting their contents in a detailed and structured format. This includes disassembling the binary to show the assembly code that the CPU executes, displaying the headers of different sections to understand the layout and structure of the binary, and listing the symbol table to see the functions and variables. It reveals how it operates and potential vulnerabilities and inefficiencies.

Utilizing objdump

The command-line nature of objdump allows me to tailor the output to my specific needs, whether investigating a particular function's assembly code or scrutinizing the program's section headers for insights into its memory layout. For example, examining the .text section for executable instructions or the .data and .bss sections for variable storage details directly impacts my ability to secure and optimize the application.

Analyzing disassembled function calls, such as those in my convert_address function, I can see the compiler's optimization at work. This aids in understanding the application's efficiency and identifying any unnecessarily complex call sequences that might introduce security risks.

When viewed through objdump, the loops within my receive_files function can reveal inefficiencies or redundancies. Moreover, the cost of branching, crucial in handling client connections and data transfers, becomes apparent. Efficient and predictable branching is key to maintaining performance under potential attack scenarios.

.text Section Analysis: This section's size and content give me clues about the complexity of the executable instructions, which are directly related to the application's performance and security posture.

.data and .bss Sections: Understanding the memory footprint through these sections helps me measure the application's efficiency and potential for improvement.

.rodata Section: Insights into my application's immutable data inform decisions about the use of constants and literals, which affect memory efficiency and security.

External Symbol Reliance: Identifying external symbols, especially those marked with "U", highlights my application's dependencies on libraries or modules. This is critical to securing network applications, as reliance on external code can introduce vulnerabilities.

Exploring my server application with objdump is fundamental to securing it. By understanding the assembly-level operations, the memory layout, and the external dependencies, I'm better equipped to identify potential vulnerabilities, optimize performance, and ensure the application's robustness against threats. This deep dive into the application’s compiled binary formulates a crucial component of my approach to network security, empowering me with the knowledge to strengthen the application.

Strace

```
sami@sami-mini-pc-ubuntu:~/Documents/socket1/cmake-build-debug$ strace ./server 10.0.0.100 8080 ../../../Desktop/
execve("./server", ["./server", "10.0.0.100", "8080", "../../../Desktop/"], 0x7ffede391538 /* 45 vars */) = 0
brk(NULL)                               = 0x5af83952d000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc5a1da6c0) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7560b01df000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=64271, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 64271, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7560b01cf000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\302\211\332Pq\2439\235\350\223\322\257\201\326\243\f"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7560afe00000
mprotect(0x7560afe28000, 2023424, PROT_NONE) = 0
mmap(0x7560afe28000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7560afe28000
mmap(0x7560affbd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7560affbd000
mmap(0x7560b0016000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7560b0016000
mmap(0x7560b001c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7560b001c000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7560b01cc000
arch_prctl(ARCH_SET_FS, 0x7560b01cc740) = 0
set_tid_address(0x7560b01cca10)         = 19686
set_robust_list(0x7560b01cca20, 24)     = 0
rseq(0x7560b01cd0e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7560b0016000, 16384, PROT_READ) = 0
mprotect(0x5af8390a3000, 4096, PROT_READ) = 0
mprotect(0x7560b0219000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7560b01cf000, 64271)           = 0
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
getrandom("\x57\x2b\xde\x33\xe1\x70\x8b\xa0", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x5af83952d000
brk(0x5af83954e000)                     = 0x5af83954e000
write(1, "Binding to: 10.0.0.100:8080\n", 28Binding to: 10.0.0.100:8080
) = 28
bind(3, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("10.0.0.100")}, 16) = 0
write(1, "Bound to socket: 10.0.0.100:8080"..., 33Bound to socket: 10.0.0.100:8080
) = 33
listen(3, 4096)                         = 0
write(1, "Listening for incoming connectio"..., 38Listening for incoming connections...
) = 38
rt_sigaction(SIGINT, {sa_handler=0x5af8390a0e4c, sa_mask=[], sa_flags=SA_RESTORER, sa_restorer=0x7560afe42520}, NULL, 8) = 0
poll([{fd=3, events=POLLIN}], 1, -1)    = 1 ([{fd=3, revents=POLLIN}])
```

Strace operates by intercepting and logging the system calls invoked by a process and the signals it receives. This is fundamentally rooted in the ptrace system call, allowing strace to monitor the calls between my application and the Linux kernel. By examining these calls, including their arguments, return values, and execution time, strace provides a transparent view of the application's runtime behaviour.

In using strace, I focused on capturing and analyzing the system calls made by my server application during its initiation and operational phases. This involves starting strace with my server application as the target, capturing a comprehensive log of all interactions with the kernel. My objective here is to ensure optimal performance and scrutinize these interactions for potential security weaknesses in how the application handles memory, manages file descriptors, or establishes network connections.

The execve system call, which marks the start of the application is critically examined to ensure that the execution environment is secure. Following this, the dynamic linking of libraries (e.g., libc.so.6) is analyzed to ensure that secure, up-to-date versions of these libraries are used, mitigating the risk of exploiting known vulnerabilities.

The operations for memory allocation (brk, mmap) are scrutinized for patterns that could indicate vulnerabilities such as buffer overflows or improper access controls. Effective memory management prevents attacks that target the application's memory space.

The setup of network communication through socket, bind, listen, and accept calls is of paramount interest. I ensure these operations adhere to best security practices, such as using appropriate socket options and binding to secure ports, to fend off unauthorized access attempts or denial-of-service attacks.

How incoming data is processed (via openat, read, write, close) offers crucial insights into potential points of data injection or inadequate validation. Ensuring robust handling of external inputs safeguards the application against various data-related attacks.

Lastly, the cleanup process, particularly the closing of resources, is evaluated for its thoroughness. Proper resource management, including the secure closure of file descriptors and network sockets, is essential to maintaining the application's security integrity over time.

In summary, my exploration of strace has enhanced my understanding of underlying system interactions. This approach has informed my optimization efforts and enabled me to identify and address potential security vulnerabilities, thus strengthening the security posture of my server applications.

Ltrace

```
sami@sami-mini-pc-ubuntu:~/Documents/socket1/cmake-build-debug$ ltrace ./server 10.0.0.100 8080 ../../../Desktop/
getopt(4, 0x7ffee6c36978, "h")                  = -1
__errno_location()                              = 0x72c6010456c0
strtoumax(0x7ffee6c3845f, 0x7ffee6c35708, 10, 0x7ffee6c35708) = 8080
__errno_location()                              = 0x72c6010456c0
memset(0x7ffee6c357d0, '\0', 128)               = 0x7ffee6c357d0
inet_pton(2, 0x7ffee6c38454, 0x7ffee6c357d4, 0x7ffee6c357d0) = 1
socket(2, 1, 0)                                 = 3
setsockopt(3, 1, 2, 0x7ffee6c35774)             = 0
inet_ntop(2, 0x7ffee6c357d4, 0x7ffee6c35710, 46) = 0x7ffee6c35710
printf("Binding to: %s:%u\n", "10.0.0.100", 8080Binding to: 10.0.0.100:8080
) = 28
htons(8080, 0x5eb79c2c32a0, 0, 1)               = 0x901f
bind(3, 0x7ffee6c357d0, 16, 0x7ffee6c357d0)     = 0
printf("Bound to socket: %s:%u\n", "10.0.0.100", 8080Bound to socket: 10.0.0.100:8080
) = 33
listen(3, 4096, 4096, 1)                        = 0
puts("Listening for incoming connectio"...Listening for incoming connections...
)      = 38
memset(0x7ffee6c356b0, '\0', 152)               = 0x7ffee6c356b0
sigemptyset(<>)                                 = 0
sigaction(SIGINT, { 0x5eb79b6e0e4c, <>, 0, 0 }, nil) = 0
realloc(0, 8)                                   = 0x5eb79c2c36b0
poll(0x5eb79c2c36b0, 1, 0xffffffff, 1)          = 1
__errno_location()                              = 0x72c6010456c0
accept(3, 0x7ffee6c356c0, 0x7ffee6c356b8, 0x7ffee6c356c0) = 4
puts("New connection established"New connection established
)              = 27
realloc(0, 4)                                   = 0x5eb79c2c36d0
realloc(0x5eb79c2c36b0, 16)                     = 0x5eb79c2c36b0
poll(0x5eb79c2c36b0, 2, 0xffffffff, 2)          = 1
read(4, "\005", 4)                              = 4
printf("\nreceiving files from client %d\n"..., 1
receiving files from client 1
) = 31
read(4, "1.png", 5)                             = 5
read(4, "\027<", 4)                             = 4
snprintf("../../../Desktop//1.png", 1024, "%s/%s", "../../../Desktop/", "1.png") = 23
fopen("../../../Desktop//1.png", "wb")          = 0x5eb79c2c36f0
printf("File name: %s with the File size"..., "1.png", 15383File name: 1.png with the File size: 15383 is receiving.
) = 57
read(4, "\377\003", 4)                          = 4
malloc(1023)                                    = 0x5eb79c2c38d0
memset(0x5eb79c2c38d0, '\0', 1023)              = 0x5eb79c2c38d0
read(4, "\377\330\377\340", 1023)               = 1023
fwrite("\377\330\377\340", 1, 1023, 0x5eb79c2c36f0) = 1023
fflush(0x5eb79c2c36f0)                          = 0
free(0x5eb79c2c38d0)                            = <void>
read(4, "\377\003", 4)                          = 4
malloc(1023)                                    = 0x5eb79c2c38d0
memset(0x5eb79c2c38d0, '\0', 1023)              = 0x5eb79c2c38d0
read(4, "\300~\204MO\365/;J*\215-JvE\t\354\265\322\a\331C\2136\211\211\\\207\210uoh"..., 1023) = 1023
fwrite("\300~\204MO\365/;J*\215-JvE\t\354\265\322\a\331C\2136\211\211\\\207\210uoh"..., 1, 1023, 0x5eb79c2c36f0) = 1023
fflush(0x5eb79c2c36f0)                          = 0
```

ltrace is a diagnostic tool that intercepts and records the library calls made by a program and the signals it receives. Unlike strace, which monitors system calls to the kernel, ltrace focuses on the calls made to shared libraries, providing a higher-level view of a program's behaviour. This is crucial in network security, where understanding the interaction between application code and external libraries can identify vulnerabilities or inefficient use of library functions. In essence, ltrace helps to examine an application's interaction patterns and dependencies, shedding light on how external code contributes to its overall functionality and potential security posture.

ltrace has allowed me to observe the execution flow in detail. For instance, the initial call to getopt reveals the parsing of command-line arguments, an essential step in configuring the server's runtime environment.

Subsequent calls, such as __errno_location, strtoumax, and memset, highlight the application's setup tasks, including error handling and memory initialization. Understanding these operations is critical in assessing the application's resilience to common attack vectors, such as memory corruption or improper error handling, which attackers can exploit.

The interaction with network-related library functions, such as socket, setsockopt, bind, listen, and accept, is particularly interesting. These functions lay the groundwork for the server's network communication capabilities. Observing these calls through ltrace provides insights into how the application establishes its networking environment, an area often targeted by attackers. For example, the configuration options set with setsockopt can affect the server's vulnerability to denial-of-service attacks or other network-based exploits.

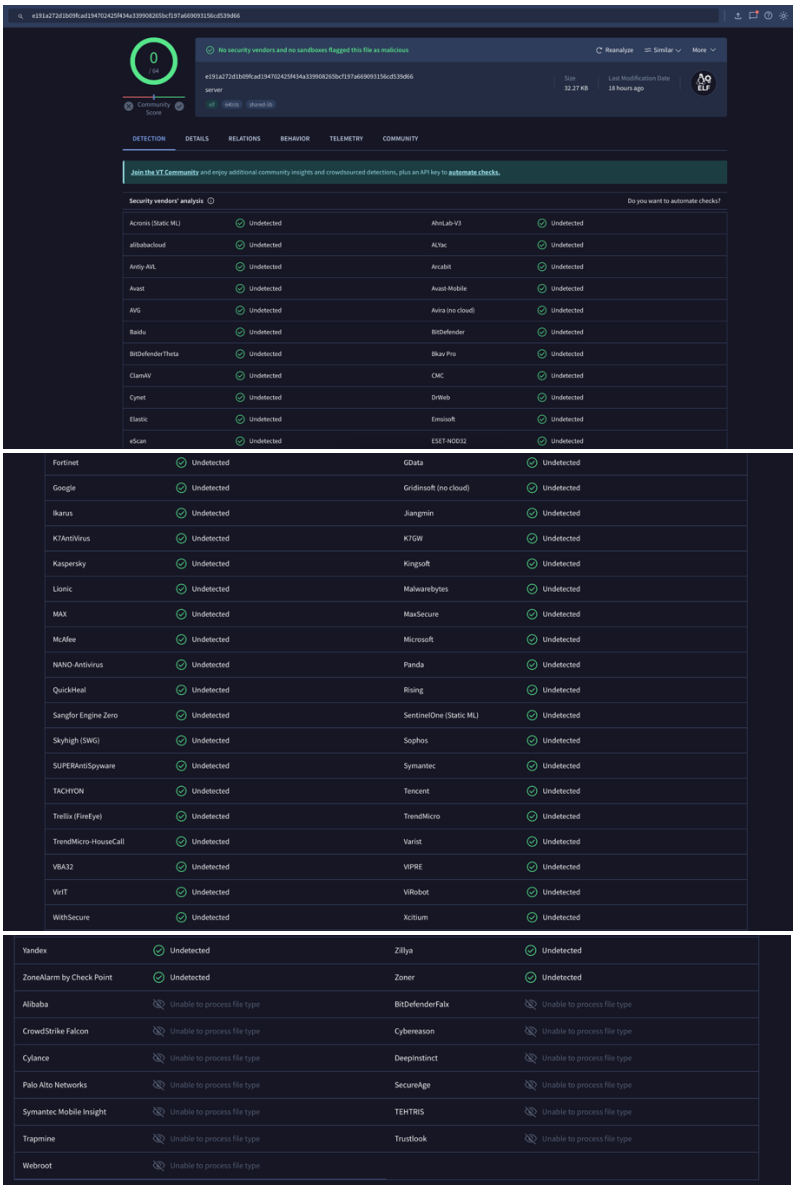
Additionally, the output related to file handling operations, signified by calls to fopen, fwrite, and fflush, is crucial in understanding how the server processes and stores data received over the network. In network security, ensuring these operations are performed securely and efficiently is paramount to prevent vulnerabilities such as arbitrary file write or disclosure of sensitive information.

Analyzing the ltrace output f has reinforced the importance of analysing the interaction between my server application and its linked libraries. This detailed examination helps identify potential security flaws, optimize performance, and ensure the application adheres to best practices in secure coding and library usage.

Furthermore, Ltrace allows me to peer into the library-level operations of my server application; I've gained a deeper understanding of its runtime behaviour and potential security implications.

In conclusion, the insights from Ltrace extend to form a foundation for building a deeper understanding of application security in a networked environment. This approach aligns with advancing the security and integrity of network applications, emphasizing a proactive and informed stance towards software development in the field of network security.

VirusTotal



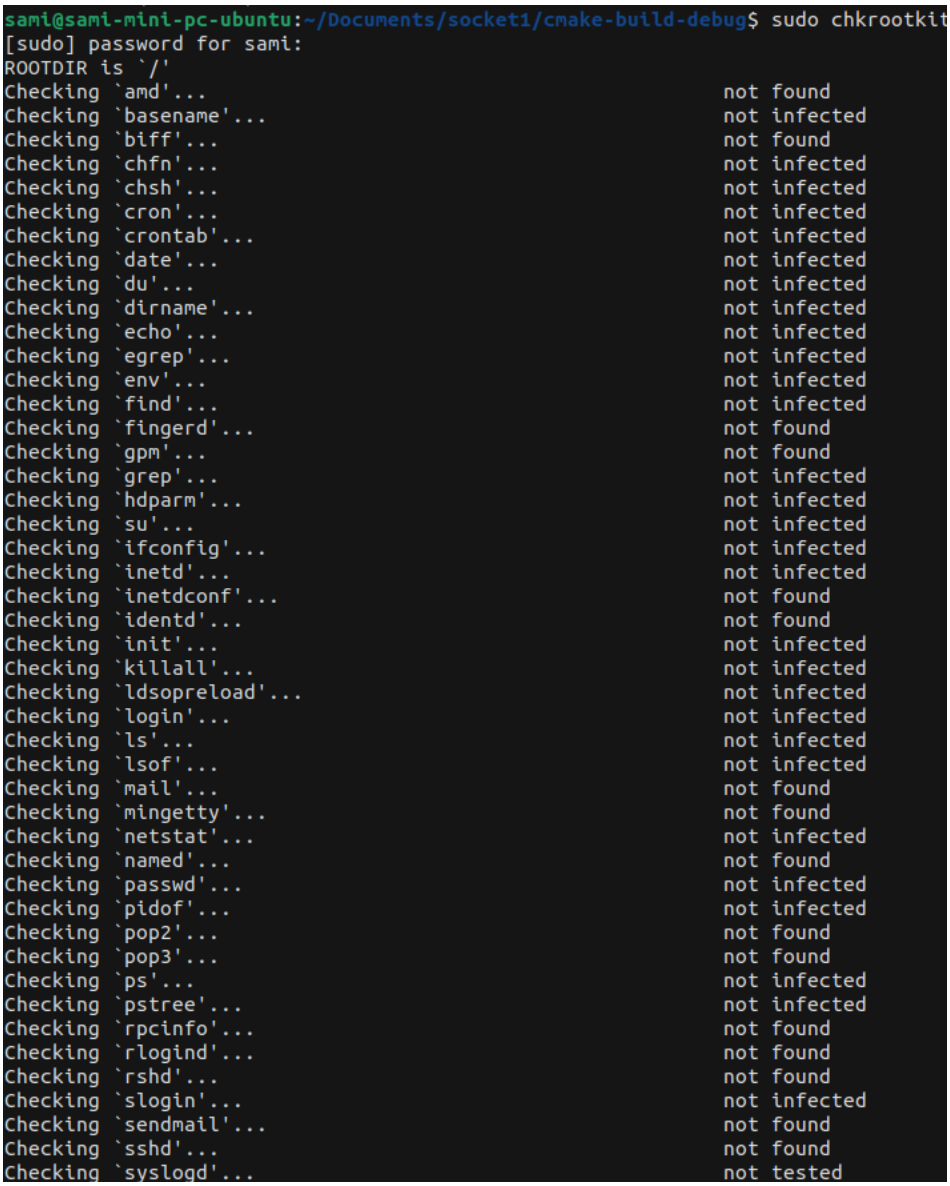
VirusTotal is a service that analyzes suspicious files and URLs to detect malware. It aggregates multiple antivirus products and online scan engines to check for viruses that the user's antivirus may have missed or to verify against false positives.

I uploaded my file through their website or via an API. This tool is invaluable in network security as it allows me to ensure that any software is free from malware and, thus, safe to use. Upon submitting a file or URL, VirusTotal scans it with many antivirus scanners and reports on what each one finds. A comprehensive report is generated that tells me whether any antivirus

engines detected the file as malicious, and it provides specific details like what type of malware was detected.

The output from the scans I initiated reveals that my file was not detected as malicious by any of the 64 antivirus engines currently used by VirusTotal. This indicates that the file is most likely safe. However, in the context of network security, no detection does not unequivocally mean the file is harmless. A new or highly sophisticated piece of malware can evade detection. Therefore, this is one of several steps in a broader security strategy, including sandboxing the file, analyzing its behaviour, and checking for any potential security-related updates or patches.

Chkrootkit



Chkrootkit is a tool designed to stealthily scan for rootkit signatures on Unix-like systems. It is a powerful utility that scrutinizes system binaries and behaviours for anomalies that may indicate a system compromise. The process requires elevated permissions; hence, I execute it with sudo, granting it access to inspect the system files.

chkrootkit methodically probes known locations and common rootkit hiding spots, checking command binaries and system behaviours against a database of known rootkit fingerprints. This includes analyzing startup scripts, network interfaces, and system logs for anomalies that typically evade notice.

The output from chkrootkit provides a report on the system's health regarding rootkit infection. As it enumerates through checks, the program categorically identifies each item as 'not infected', 'not found', or 'not tested'. For instance, when it labels a binary like 'chfn' as 'not infected', it reassures me that this binary hasn't been modified to hide a malicious presence. When it reports 'not found', it suggests those services or files don't exist on the system, either because they've never been installed or are not part of the standard distribution on my system.

The utility also flags 'suspicious files and directories'—a call to action for me to scrutinize these areas more closely. While false positives are common, these flags are a starting point for a deeper dive into the system's integrity. Furthermore, when inspecting network interfaces, chkrootkit seeks out signs of promiscuity, which can hint at a sniffer's presence, potentially leaking sensitive information.

The output provides reassurances and calls for observation. For example, the report about processes without entries in the utmp file could reflect the nature of chkrootkit's operation or point towards a stealthier manipulation.

In the broader network security and system installation context, chkrootkit represents an essential first line of defense. The output from chkrootkit, especially during the installation phase, informs me whether the system is secure, or I must take action to ensure security. It acts as an initial security state, offering peace of mind or signalling the need for further investigation before proceeding with additional network security implementations.

Conclusion

After examining various security and analysis tools during the network installation phase, I've established a comprehensive overview of the network's current security posture. Strings allowed me to extract human-readable content from binaries, shedding light on the server program's functionality and potential security vulnerabilities. The nm tool was instrumental in mapping out symbols within the compiled program, which provided critical insights into the program's behaviour, dependencies, and architectural underpinnings.

Objdump facilitated a deep dive into the program's binary structure, unveiling the assembly code vital for identifying potential inefficiencies and vulnerabilities. Strace and ltrace proved invaluable in tracing system and library calls, respectively, offering a transparent view of the application's interactions with the operating system and its use of shared libraries. This was crucial for pinpointing weaknesses and ensuring compliance with secure coding practices.

VirusTotal was an essential tool in scanning files for malware, ensuring the network is free from known threats. Although no threats were detected, I acknowledge the need for continuous vigilance, considering the evolving nature of cybersecurity threats. Chkrootkit reinforced the first line of defence, signalling a clear system state and highlighting areas that require further analysis.

Moving forward, I recommend a strategy that combines ongoing surveillance with the proactive updating of defence mechanisms. Regular audits using these tools, coupled with updates and patches to the system, will be crucial in adapting to new threats. Training for users on best security practices and staying informed about the latest security developments will also be integral to enhancing network security. Implementing these recommendations will strengthen our network's defenses and maintain the integrity and trustworthiness of our digital infrastructure.

Appendix (Outputs)

All the outputs are inside the data folder in Txt format.