

HOOKING

Potato ACADEMY

Vulnerability research

- ❖ Compilation Phases
- ❖ Linking
- ❖ Static vs Dynamic Linking
- ❖ Hooking

Compilation Phases

Preprocessing – generates macro code, removes comments, searches for included header files etc

Compiling – turns the code into assembly

Assembly – turns the assembly code into binary.
This is known as object code (hence *.o files)

Linking – combines the object files from assembly
into a single file, links calls from shared libraries
Static or dynamic linking

Static Linking

- ❖ Takes the functions from shared libraries and copies the code into the output binary
- ❖ To generate statically linked binaries, add the “-static” flag when compiling with gcc
- ❖ What would be the result of compiling this code statically?
- ❖ This code has 2 calls to external libraries –
 - ❖ atoi
 - ❖ printf

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *c = "1";
    int i = atoi(c);

    printf("%d\n", i);
}
```

Static Linking

- ❖ The “file” command will tell you that it is statically or dynamically linked
- ❖ Note that the file size is 695k

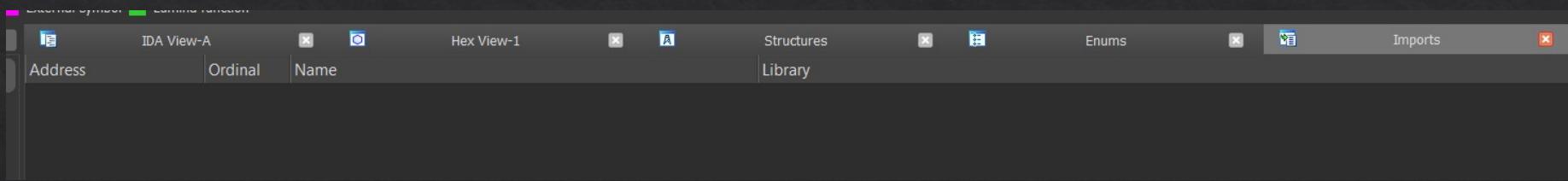
```
(kali㉿kali)-[~/vr_lessons/link]
$ gcc -m32 linking.c -o linking-static -static

(kali㉿kali)-[~/vr_lessons/link]
$ file linking-static
linking-static: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically linked, BuildID[sha1]=a
ba4f159e87a98e857d40c45ce5ab3489de5b054, for GNU/Linux 3.2.0, not stripped

(kali㉿kali)-[~/vr_lessons/link]
$ ls -lah linking-static
-rwxr-xr-x 1 kali kali 695K Nov 27 01:05 linking-static
```

Static Linking

- ❖ As the linker copies relevant external functions into the binary, there is nothing in the Imports tab in IDA



- ❖ Code for atoi is present in the binary. Other dependent functions like strtol will also be present

```
public atoi
atoi proc near

arg_0= dword ptr 4

; __unwind {
push ebx
call _x86_get_pc_thunk_bx
add ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
sub esp, 0Ch
push 0Ah
push 0
push [esp+18h+arg_0]
call strtol
add esp, 18h
pop ebx
retn
; } // starts at 804FBF0
atoi endp
```

The assembly code for the atoi function is shown in the IDA Pro assembly view. It is a public function named atoi. The code begins with a prologue that pushes the current base pointer (ebx) onto the stack, calls _x86_get_pc_thunk_bx to get the address of the global offset table, adds it to ebx, and then subtracts the size of the table from esp. It then pushes 0Ah and 0 onto the stack, pushes the address of arg_0 onto the stack, and calls the strtol function. After the call, it adds 18h to esp, pops ebx off the stack, and returns. A note at the bottom indicates the code starts at address 804FBF0.

Dynamic Linking

- ❖ Default setting if “-static” is not supplied when compiling
- ❖ Links external functions and doesn’t copy them into the binary
- ❖ Note that the file size is 15K compared to 695K when statically linked

```
(kali㉿kali)-[~/vr_lessons/link]
$ gcc -m32 linking.c -o linking-static

(kali㉿kali)-[~/vr_lessons/link]
$ file linking
linking: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=01edcb154f3df034acecdb2711e16d30624441cd, for GNU/Linux 3.2.0, not stripped

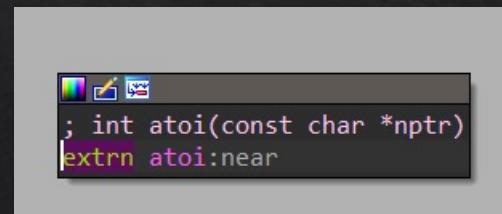
(kali㉿kali)-[~/vr_lessons/link]
$ ls -lah linking
-rwxr-xr-x 1 kali kali 15K Nov 27 01:05 linking
```

Dynamic Linking

- ❖ Imports table in IDA shows import table with the linked functions

Address	Ordinal	Name	Library
00004028		printf@@GLIBC_2.0	.dynsym
0000402C		_cxa_finalize@@GLIBC_2.1.3	.dynsym
00004030		_libc_start_main@@GLIBC_2.0	.dynsym
00004034		atoi@@GLIBC_2.0	.dynsym
00004038		_ITM_deregisterTMCloneTable	.dynsym
0000403C		_gmon_start_	.dynsym
00004040		_ITM_registerTMCloneTable	.dynsym

- ❖ Code for atoi doesn't exist in the binary



```
; int atoi(const char *nptr)
extrn atoi:near
```

STATIC LINKING

VERSUS

DYNAMIC LINKING

STATIC LINKING

Process of copying all library modules used in the program into the final executable image

Last step of compilation

Statistically linked files are larger in size

Static linking takes constant load time

There will be no compatibility issues with static linking

DYNAMIC LINKING

Process of loading the external shared libraries into the program and then bind those shared libraries dynamically to the program

Occurs at run time

Dynamically linked files are smaller in size

Dynamic linking takes less load time

There will be compatibility issues with dynamic linking

Static vs Dynamic

- ❖ <https://pediaa.com/what-is-the-difference-between-static-and-dynamic-linking/>
- ❖ <https://medium.com/@birnbera/static-vs-dynamic-libraries-5912efe9bf52>

Dynamic Linking

- ❖ Dynamic linking allows us to hook externally called functions and replace them with our own
- ❖ External functions are resolved during run time, when the function is called (lazy binding)
- ❖ <https://bitguard.wordpress.com/2016/11/26/an-example-of-how-procedure-linkage-table-works/>
- ❖ <https://binaryresearch.github.io/2019/08/29/A-Technique-for-Hooking-Internal-Functions-of-Dynamically-Linked-ELF-Binaries.html>

Hooking

- ❖ Why – when doing RE or fuzzing we may need to modify the behaviour of some functions
- ❖ Eg – replacing the random() call with a constant value to improve a fuzzer's stability
- ❖ Eg – replacing functions accessing hardware devices which may not be present during RE (disabling nvram_set and nvram_get calls when emulating a router's http service)
- ❖ Eg – execute a certain function in a binary as an entry point (useful for fuzzing)

Hooking Example

- ❖ upnpd binary in Netgear links to the following functions to interface with NVRAM
- ❖ In order to run upnpd in an emulated environment which does not include an NVRAM device, we will need to hook these functions
- ❖ How – by reimplementing those functions and configuring Linux to load it before the normal shared library
- ❖ LD_PRELOAD – Linux environment variable that modifies the paths to shared libraries

Symbol	Access	.dynsym...
000DBB9C	acosNvramConfig_get	.dynsym
000DB8C8	acosNvramConfig_invmatch	.dynsym
000DB8F4	acosNvramConfig_match	.dynsym
000DB9A4	acosNvramConfig_read	.dynsym
000DBA80	acosNvramConfig_save	.dynsym
000DB978	acosNvramConfig_set	.dynsym
000DB8B0	addSamainNvramDeviceValue	.dynsym

Hooking

- ❖ Reimplement functions with the same signature and return types
- ❖ Compile as a shared library eg “gcc hook.c –o hook.so –shared”
- ❖ Add “hook.so” as a value to LD_PRELOAD
- ❖ LD_PRELOAD=./hook.so /usr/sbin/upnpd
- ❖ <https://catonmat.net/simple-ld-preload-tutorial>

```
char*acosNvramConfig_get(char *key)
{
    return nvram_get(key);
}

intacosNvramConfig_set(char *key, char *value)
{
    nvram_set(key, value);
    return 0;
}

intacosNvramConfig_match(char *key, char *value)
{
    int return_value = 1;

    if (compare_keys(key, "wan_status"))
    {
        return_value = 0;
    }
    if (compare_keys(key, "wan_proto"))
    {
        if (compare_keys(value, "dhcp"))
        {
            return_value = 1;
        }
        else
            return_value = 0;
    }
    if (compare_keys(key, "httpsEnable"))
    {
        return_value = 0;
    }
    ...
    return return_value;
}
```

Hooking Example – Entry Point

- ❖ We can also modify the entry point of a binary for fuzzing or other automated testing
- ❖ Example – we may want to fuzz the method below at 0x15238

```
.text:00015238
.text:00015238
.text:00015238
.text:00015238 ; void z_handleHttpOrSambaRequest_sub_15238()
.text:00015238 z_handleHttpOrSambaRequest_sub_15238
.text:00015238
.text:00015238 var_38= -0x38
.text:00015238 anonymous_0= -0x34
.text:00015238 anonymous_1= -0x1C
.text:00015238
.text:00015238 PUSH {R4-R7,LR}
.text:0001523C SUB SP, SP, #0x1000
.text:00015240 SUB SP, SP, #0x24
.text:00015244 MOV R3, #0
.text:00015248 ADD R2, SP, #0x10038+var_38
.text:0001524C STR R3, [R2,#0x1C]
.text:00015250 LDR R0, =aLocalHttpdRequ ; "local_httpd_request_pid"
.text:00015254 LDR R1, =aUpnpd ; "upnpd"
.text:00015258 BL acosNvramConfig_match
.text:0001525C CMP R0, #0
.text:00015260 BEQ loc_15280

.text:00015264 LDR R0, =aUpnpd ; "upnpd"
.text:00015268 BL z_sub_F0A8_getPIDFile ; returns the pid stored in /var/run/<process>.pid or 0
                ;
.text:0001526C LDR R3, =dword_18AEAC
.text:00015270 MOV R2, #1
.text:00015274 MOV R4, R0
.text:00015278 STR R2, [R3,#(z_dword_18BA7C_UPNPProcessRunning - 0x18AEAC)]
.text:0001527C B loc_1528C

.text:00015280 loc_15280
.text:00015280 LDR R0, =aRemoteSmbConf ; "remote_smb_conf"
.text:00015284 BL z_sub_F0A8_getPIDFile ; returns the pid stored in /var/run/<process>.pid or 0
                ;
.text:00015284 MOV R4, R0

.text:0001528C loc_1528C
.text:0001528C LDR R5, =dword_18AEAC
                ;
.text:00015290 MOV R5, R0
```

Hooking Example – Entry Point

- ❖ When binaries are loaded, the “main” method as defined by the programmer is *not* the first method executed
- ❖ The first method being executed is the main method of the runtime C library
 - ❖ Libc for Linux
 - ❖ Uclibc for embedded Linux
 - ❖ CRT for Windows (C runtime)
- ❖ Sets up the execution environment such collecting command line args to pass to main(), initialise threading subsystems, perform security checks, etc...
- ❖ `__libc_start_main`, `__uClibc_main`...

Hooking Example – Entry Point

- ❖ We can overwrite the libc main function to change the entry point
- ❖ Seems that exact method signature does not matter. But name must match, eg for uClibc runtimes, the function name must be `_uClibc_main`
- ❖ The code below overwrites the normal `_uClibc_main` (which calls `main`) and instead calls 2 functions by addresses, `bootstrap` at `0x15494` and `handleHTTPOrSambaRequest` at `0x15238`
- ❖ When the program is compiled and run, our version of `_uClibc_main` will be called

```
void __uClibc_main(void *main, int argc, char** argv)
{
    void (*bootstrap)() = (void*)0x00015494;
    bootstrap();

    // need request in /tmp/tmp_http_request.txt
    void (*handleHTTPOrSambaRequest)() = (void*)0x00015238;
    handleHTTPOrSambaRequest();
}
```