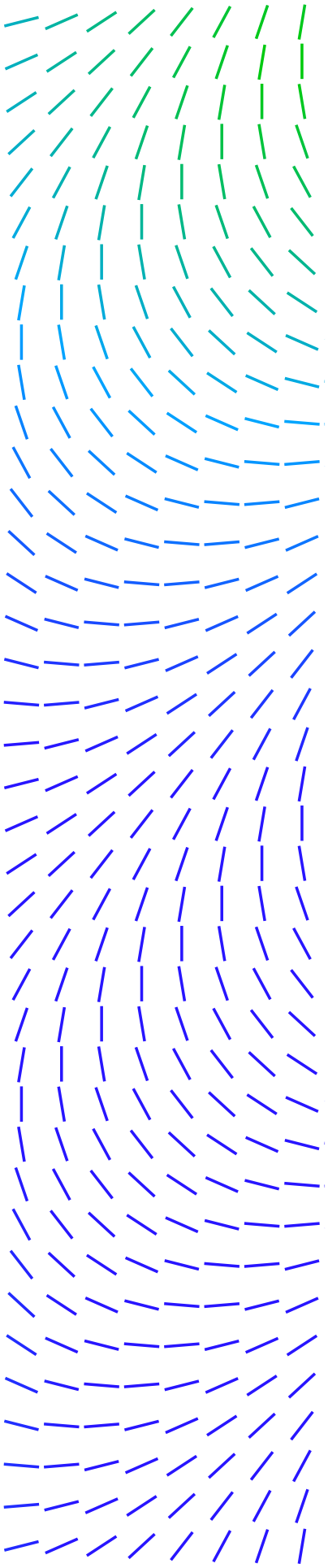

Function Hooking for Recon and Exploitation

By: Sam Quinn

Trellix

Finding vulnerabilities by reverse engineering software is often a difficult undertaking. This is especially true when attempting to understand data being passed in and out of functions. In these scenarios, dynamic debugging is often used to track and identify data moving between functions. Another option is the process of hooking function calls. The term "hooking a function" is the process of changing the default flow of execution, usually with the intent of either gathering information or changing the result of the hooked function entirely. This technique analysis will walk through three examples to help explain the process of hooking functions to retrieve useful information as well as modifying functions to a researcher's benefit.

Prerequisites:

- Linux System (a VM will work)
- Frida tools installed (how to install here)
- All code can be found here

Example #2:

- An Android Phone with root access (An emulator will work too)

Getting Started:

We will focus on a few examples highlighting the basics of function hooking as well as some real-world case studies taken from ATR's previous research. This walk through is not an in-depth tutorial on how to run the tools; instead, it details how these tools can be used to supplement the research process.

To get hands on experience with these tools and techniques, Examples #1 and #2 are designed to be followed along with.

Example 1:

In this example we will be working with a small `x86_64 ELF` binary compiled for Linux. This program is a simulated Powerball game where there are 6 random numbers chosen (equivalent to the balls in Powerball), and the user tries to guess these numbers. Depending on how many correct numbers the user chooses, they will receive a cash reward. To participate in this example, the code can be found here. This game is just a silly exercise to help teach the usefulness and power of hooking functions. With the ridiculous odds of 1 in 292,201,338 to get the jackpot, hooking functions can increase the odds of winning if the information of the winning numbers were disclosed to the user before they enter their guesses. Since this exercise is about hooking functions and using tools to instrument this process, GDB or binary patching will not be covered, but also could perform similar changes to "win."

Ultimately this led to our access violation crash. All this instrumentation and debugging data was done on the target without ever having to restart the service. Because of the nature of how this application was getting started, it did not lend itself well to being started manually, and since it was running on an Android, the `LD_PRELOAD` method of hooking would have been a major pain. Frida was able to connect to the running daemon without interrupting anything and not triggering any watchdogs. With the instrumentation Frida provided above it was easy to identify and determine the exploitability of the crash in question.

Because this is a Linux executable, with dynamic linking, generic C functions like "`rand()`" or "`printf()`" are linked to the system implementation, commonly found in libc.so installed in the OS. This linking helps reduce the size of the executable, minimizing the redundant code and can work on many different systems and versions of libc.so. However, because some of the functions are dynamically linked, we can tell the OS to use our version of a function and disregard the true implementation of these functions. This can be done by simply assigning the `LD_PRELOAD` environment variable.

To use `LD_PRELOAD` one must provide it a path to a compiled shared object with a function declaration that matches the desired function to be overwritten.

In the powerball.c code, the `rand()` function is being used to generate the winning balls. The `LD_PRELOAD` variable can be used to overwrite this function to include a printf call to disclose the winning number before the user has a chance to guess. The code shown in Figure 1, creates a `rand()` function with the same function signature as the true `rand()`. This is necessary as we are not modifying the Powerball program. The hooking code also includes code that calls the "true" `rand()` function, this isn't necessary but shows that usefulness of the `dlsym()` function for this very task. Lastly, since `LD_PRELOAD` requires a shared object file we must use specific compiler flags to turn this code into a shared object that can be imported into the Powerball game.

```c
// hook.c
// compile with: gcc -fPIC -shared hook.c -o fake_rand.so -ldl
#define _GNU_SOURCE // Needed for RTLD_NEXT
#include <stdio.h>
#include <dlfcn.h>  // Required for dlsym()

int rand(void) {
    // Create a function pointer = NULL
    int (*real_rand)(void) = NULL;
    // Populate the function pointer with the returned address from dlsym
    real_rand = dlsym(RTLD_NEXT, "rand");
    // Get a real random number from the libc rand()
    int random_number = real_rand();
    // Print the random number and return it.
    printf("Hooked Rand() returned %d\n", random_number % 100);
    return random_number;
}
```

Figure 1. Shared object code to implement "rand()"

Once the new shared object is compiled and saved into the LD_PRELOAD variable, when the game executes it will now run the modified rand() function which in turn calls the true rand() function, prints out the number, and then returns the now disclosed number, as seen in Figure 2.

```
$ LD_PRELOAD=`pwd`/fake_rand.so ./power
Ready to play? Y/N y
Hooked Rand() returned 22
Hooked Rand() returned 32
Hooked Rand() returned 35
Hooked Rand() returned 40
Hooked Rand() returned 11
Hooked Rand() returned 66
Enter guess #1: 22
Enter guess #2: 32
Enter guess #3: 35
Enter guess #4: 40
Enter guess #5: 11
Enter guess #6: 66
The winning numbers are ...  22  32  35  40  11  66
22 is a winning number
32 is a winning number
35 is a winning number
40 is a winning number
11 is a winning number
Jackpot!!
A total of 5 winning numbers plus the Powerball
Your winnings are: $40,000,000
```

Figure 2. Output from the power game after using LD_PRELOAD with our modified "rand()" function.

As seen in Figure 2, the `LD_PRELOAD` environment variable is useful and allowed the user to see each number as it was randomly generated. The downside of this is that each hooked function call must be compiled into a "`.so`" object and have the target application spawned with the `LD_PRELOAD` variable present. Lastly the target application must be dynamically linked or the `LD_PRELOAD` variable would be ignored as no functions are being looked up.

With the basics of how hooking can be completed manually we can introduce dynamic instrumentation. For this walk through we will focus on Frida and how it can overcome the issues presented by manually hooking function calls. Frida is a very powerful tool that can be used similar to `LD_PRELOAD` but can overcome many of the limitations of the `LD_PRELOAD` method. Frida can attach to running programs and hook functions dynamically as well as functions from a static binary.

To achieve exactly as we did before with `LD_PRELOAD`, the Frida code, written in JavaScript, is much simpler using Frida's Interceptor functionality.

```
'use strict';

Interceptor.attach(DebugSymbol.fromName("rand").address, {
    onLeave: function (retval) {
        console.log("RetVal: " + retval.toInt32() % 100);
        console.log("Replacing RetVal with: 12");
        retval.replace(12);
    }
});
```

Figure 3. Frida code to hook, print the value, and replace the return value of "rand()"

The code in Figure 3 tells Frida on exiting the "`rand()`" function to print the return value and then replace the return value with 12. The replacing the return value is just to show the flexibility Frida has with its function hooking abilities.

Now, to hook the function using the Frida script from Figure 3, the application can be started normally. As seen in Figures 4 and 5, before typing "y" to play, start Frida and pass the PID of the power executable and the script to Frida.

```
Ready to play? Y/N
y
Enter guess #1: 12
Enter guess #2: 12
Enter guess #3: 12
Enter guess #4: 12
Enter guess #5: 12
Enter guess #6: 12
The winning numbers are ...  12  12  12  12  12  12
12 is a winning number
12 is a winning number
12 is a winning number
12 is a winning number
12 is a winning number
Jackpot!!
A total of 5 winning numbers plus the powerball
Your winnings are: $40,000,000
```

Figure 4. Starting the game normally and hooking it with Frida

```
$ frida -p 8407 -l ./hook_rand.js
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/

[Local::PID::8407]->
RetVal: 95
Replacing RetVal with: 12
RetVal: 19
Replacing RetVal with: 12
RetVal: 42
Replacing RetVal with: 12
RetVal: 77
Replacing RetVal with: 12
RetVal: 20
Replacing RetVal with: 12
RetVal: 12
Replacing RetVal with: 12
Process terminated
[Local::PID::8407]->

Thank you for using Frida!
```

Figure 5. Hooking the power process with Frida and passing in a Frida script

It almost seems like magic, how Frida transparently modifies the process to disclose the original random numbers and replace all with the value 12. While we passed in our Frida script, Frida can also act as an interpreter, which allows commands to be entered directly into the Frida "shell" to perform all the dynamic instrumentation needs.

Example 2:

For a more realistic example, we can use Frida to hook functions within an Android application to disclose sensitive information. This example also highlights the cross-compatibility Frida has with the platform-agnostic functionality of the hooking framework. When reverse engineering Android applications, it is often hard to follow code flow or determine how variables are populated. One example is the temporary credentials returned by Amazon's AWS Cognito services. The Cognito services provide user access to AWS resources, usually for a limited time. Sometimes these credentials have more access than needed for default functionality and can give a researcher a new attack vector to AWS resources. Check out our blog here on the iParcelBox, where we were able to use the AWS Cognito credentials to access every iParcelBox deployed. While Frida can orchestrate the unpinning of SSL certs to potentially show the returned AWS Cognito credentials from a network capture, it can sometimes be easier to just use Frida to hook the function, and have it print the credentials to the console. To follow along in this example it will be left to the reader to find an Android application that uses AWS Cognito services; there are many of them out there, including the iParcelBox app we investigated. The hooked java classes in Figure 6 will potentially need to be adjusted as well.

The IdToken and AccessToken are both provided dynamically from the AWS Cognito server and would require an exhaustive reverse engineering approach to uncover the correct calls and destination URL statically.

Yet, by simply hooking these functions and printing the return value as shown in Figure 6, we can use the application to log in to the Android application normally, and have these credentials nicely printed to the console as seen in Figure 7.

The possibilities of information that can be gathered from dynamic hooking provided by Frida is limitless. A few other interesting applications of using Frida to hook within Android applications could be to gather an encryption key when an encryption function is called or hook a video game function to change the health impact of an attack. Each of these could be done without having to modify the Android applications at all.

```
console.log("Hooking Script loaded");
Java.perform(function x() {
    // Find the class you want to hook, Use Jadx-gui to find the class
    var my_class = Java.use("com.amazonaws.mobile.client.results.Tokens");
    // Overwrite the function with our custom one
    my_class.getIdToken.implementation = function () {
        console.log("Call to getIdToken");
        // Get the results of the "real" or original function
        var ret_value = this.getIdToken();
        // Print the results of the function
        console.log("IdToken: " + ret_value.getTokenString());
        return ret_value;
    }
    // Do this for as many functions as you like
    my_class.getAccessToken.implementation = function () {
        console.log("Call to getAccessToken");
        var ret_value = this.getAccessToken();
        console.log("AccessToken: " + ret_value.getTokenString());
        return ret_value;
    }
});
```

Figure 6. Frida code to hook the AWS Cognito Token functions on Android

```
Call to getIdToken
IdToken:
eyJraWQiOiI2RkJ2SFhcL0Z1UTlcL1BxUFRpU0NQTXZ5U1ZCNG1KcndhVkt4ZU45eTdzckE9IiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiI1YmMxNT
k0Yi0yMGMxLTQxMDItYmY5MC1iZTk1YzFmNzRkNTgiLCJlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiY3VzdG9tOlN0cmlwZV9JRCI6ImN1c19LUmI4SGl
2bFF1U0txcSIsImlzcyI6Imh0dHBzOlwvXC9jb2duaXRvLWlkcC5ldS13ZXN0LTEuYW1hem9uYXdzLmNvbVwvZXUtd2VzdC0xX1pUQnl5RUFwWCIs
ImNvZ25pdG86dXNlcm5hbWUiOiI1YmMxNTk0Yi0yMGMxLTQxMDItYmY5MC1iZTk1YzFmNzRkNTgiLCJnaXZlbl9uYW1lIjoiQm9iIiwiYXVkIjoiM
Ws5Y200YXUzdTZqNmExMDByMTdkOWRjaTgiLCJ0b2tlbl91c2UiOiJpZCIsImF1dGhfdGltZSI6MTYzNDg1MDk4MSwiZXhwIjoxNjM0ODU0NTgxLC
JpYXQiOjE2MzQ4NTA5ODEsImZhbWlseV9uYW1lIjoiQm9iIiwiZW1haWwiOiJpMnA0cThxaG12YmdAb3BheXEuY29tIn0.MaGKbMaePIJDZdlEZ9B
1nkPYfSdG24LAZ9FG5O4V_FIGT1gJYK4Mx160Ugt4DPGMShxYkIXQOHilmrNm7Q_ztAW_3wdrwduZS2BgRmOW0E6vUc1Big8z3RyU337GGAPSJ8fM
z5hKWNM9Xb7ip3nim1YtKPc_oFUcBMLUhQo1FvRJ5t8syh1IyRQdwvGtjL20wsfVlOuB9NR4TZi_jimeO4LIithdqSoQgyxG_HlsUBPZs4SGlRLdk
bsgAAZ8Wx-GtgJCFH4dBEraGCgDHbcuNjhV-MrGhJ9E-zLzp9mMgszAJxyHXvC54Q2lYPBiX05Oa3dBWCQadsXje-oCw

Call to getAccessToken
AccessToken:
eyJraWQiOiJuelZ5dDdOcDlcL0xEUXNVREFmM2VSdUlkZ1grOG9lTCtud3h2MnhOa05RTT0iLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiI1YmMxNTk0
Yi0yMGMxLTQxMDItYmY5MC1iZTk1YzFmNzRkNTgiLCJkZXZpY2Vfa2V5IjoiZXUtd2VzdC0xXzUwM2MzMmJmLTZhZTgtNGRkYi1iMzYwLTM0NjNlM
GNkNGQwNiIsInRva2VuX3VzZSI6ImFjY2VzcyIsInNjb3BlIjoiYXdzLmNvZ25pdG8uc2lnbmluLnVzZXIuYWRtaW4iLCJhdXRoX3RpbWUiOjE2Mz
Q4NTA5ODEsImlzcyI6Imh0dHBzOlwvXC9jb2duaXRvLWlkcC5ldS13ZXN0LTEuYW1hem9uYXdzLmNvbVwvZXUtd2VzdC0xX1pUQnl5RUFwWCIsImV
4cCI6MTYzNDg1NDU4MSwiaWF0IjoxNjM0ODUwOTgxLCJqdGkiOiJmMWYwM2M0OS05OGU4LTQ1MzEtYmFkOC04ZjYzYjQ2MDUxNTYiLCJjbGllbnRf
aWQiOiIxazljbTRhdTN1Nmo2YTEwMHIxN2Q5ZGNpOCIsInVzZXJuYW1lIjoiNWJjMTU5NGItMjBjMS00MTAyLWJmOTAtYmU5NWMxZjc0ZDU4In0.P
lJlTm7l19kjnxXH_mMB-ahSRx31tNA3cm5EazHFkAlKl4UXanZ4jH__Yo5j_Cb4pAKZAf9JPYZeCo0LTKp_uIF_S7oWWi3bTyejMEZowIGGYUjV-
qiXBzI15puoh39Wk8D1Tr94XGycv8hYaTv3JX4YTLwxcG7QQNsl90fHOrU2_H2g4ZTQHqVGrJwZvurOzCve7rTlya3dtvfYt9LI0ghbJNYny65D3P
ibonvcP1EpV7oDZtaVICMD2AwUBRdQIVldcxs4uVewCXca87UY0RZ1HcjOXbiFsPLvUkQVc9potYxdPXR7ilwNPwPitGPqzqS--KqSabhrApvU20g
```

Figure 7. Trimmed down output from the hooked AWS Cognito token functions

Example 3:

For the last example, Frida will be used to help narrow down a crash and determine the root cause. This is another real example from our team's research into the Peloton and later assigned CVE-2021-40526. You can read more about this research here. In short, there was a read access violation that would occur only after sending more than one network packet to the target. The team's original thought was that it may have been a use after free vulnerability. Frida was used to track each `malloc()` and `free()` call to see where the memory address in question was eventually leading to an access violation. This example will be extremely hard to follow along with since after we reported this to Peloton they patched the system to no longer allow users to get root access to the device. But this example highlights the power that Frida has to help narrow down crashes and touches on some advanced functionality.

We used the following Frida code in Figure 8 to keep track of each malloc and free call as well as the addresses they are tied to and nicely print the info to the console.

```
Interceptor.attach(DebugSymbol.fromName("malloc").
  onEnter: function (args) {
    console.log("\n----Malloc----")
    console.log("Size : " + args[0])
    console.log("LR :" + DebugSymbol.fromAddress(t
    var socket_head = DebugSymbol.fromName("s_sock
  },
  onLeave: function (retval) {
    console.log("Address: " + retval)
  }
});

Interceptor.attach(DebugSymbol.fromName("free").ad
  onEnter: function (args) {
    console.log("\n----Free----")
    console.log("Address : " + args[0])
    console.log("LR :" + DebugSymbol.fromAddress(t
  },
});
```

Figure 8. Frida code to track each "malloc()" and "free()" call to determine if the crash was a use after free vulnerability

Hooking the target process in question and sending our crash case resulted in the following output shown in Figure 9.

```
Address : 0x733e25a020
LR :0x5819d085ac gymkitDaemon!m_recvDataCB vendor/peloton/platform_apps/GymkitDaemon/gymkitDaemon.c:862

----Malloc----
Size : 0x10037
LR :0x5819d055e4 gymkitDaemon!__iAP2BuffPoolGetSendPacket vendor/peloton/platform_apps/GymkitDaemon
/iAP2UtilityImplementation/iAP2BuffPoolImplementation.c:184
Address: 0x733e2b24c0

----Malloc----
Size : 0x10037
LR :0x5819d055e4 gymkitDaemon!__iAP2BuffPoolGetSendPacket vendor/peloton/platform_apps/GymkitDaemon
/iAP2UtilityImplementation/iAP2BuffPoolImplementation.c:184
Address: 0x733e2c75c0

----Free----
Address : 0x733e25b000
LR :0x5819d0559c gymkitDaemon!__iAP2BuffPoolReturnBuff vendor/peloton/platform_apps/GymkitDaemon
/iAP2UtilityImplementation/iAP2BuffPoolImplementation.c:133

{
  "message": "access violation accessing 0xe8fe60173b26033a",
  "type": "access-violation",
  "address": "0x5819d091c8",
  "memory": {
    "operation": "read",
    "address": "0xe8fe60173b26033a"
  }
```

Figure 9. Frida output showing each malloced address and each address being freed

We noticed that the access violation was coming from what looks like a random address, which would change every run. The next step was to track the data structures within the malloced memory. The data structure of interest was a doubly linked list that kept track of each socket connection. Each socket was in a structure as seen in Figure 10.

```
typedef struct _NODE
{
    int                 socket_fd;
    unsigned long long  lasttick;
    FILE*               filehandle;
    struct _NODE*       prev;
    struct _NODE*       next;
}NODE;
```

Figure 10. The data structure for each network connection

To view this information systematically in Frida, the data structure can be accessed with the Frida code from Figure 11. Because the target application in this case had symbols, we were able to use the function names within our Frida code. Once the first item in the linked list address is returned from the "`List_NetGetFirstNode()`" function, we can loop through the linked list and print the values at each of the structs members to disclose useful information.

```
Interceptor.attach(DebugSymbol.fromName("List_NetGetFirstNode").address, {
    onLeave: function (retval) {
        if (glob_sockethead == "0x1337") {
            console.log("SETTING GLOB PTR TO: " + retval)
            globalThis.glob_sockethead = ptr(retval)
        };
        var s_sockethead = retval;
        while (true) {
            console.log("\ns_sockethead: " + s_sockethead)
            console.log("-- socketfd: " + ptr(s_sockethead).readPointer())
            console.log("-- prev: " + ptr(s_sockethead).add(0x10).readPointer())
            console.log("-- next: " + ptr(s_sockethead).add(0x18).readPointer())
            if (ptr(s_sockethead).add(0x18).readPointer() == "0x0") {
                break;
            } else {
                s_sockethead = ptr(s_sockethead).add(0x18).readPointer();
            }
        }
    },
});
```

Figure 11. Frida code to print off each connection data structure from the linked list

The "`List_NetGetFirstNode()`" function was called each time a write to the linked list would occur and seemed to be a suitable candidate for instrumentation. The results can be viewed in Figure 12.

```
s_sockethead: 0x733e25a000
-- socketfd: 0x5
-- prev: 0x0
-- next: 0x733e25a020

s_sockethead: 0x733e25a020
-- socketfd: 0x7
-- prev: 0x733e25a000
-- next: 0x0
```

Figure 12. Frida output showing the useful info from the connection data structure

This shows that there are two active connections and that they are pointing to each other. With this information, the crash condition was again sent to the target, and it was clear that at a certain point the "`socketfd`" seems to get a weird value much greater than what is considered normal for a file descriptor as seen in Figure 13.

After sending a few more packets to the target, the entire structure (including the previous and next pointers) seems to get messed up:

```
s_sockethead: 0x733e25a000
-- socketfd: 0x5
-- prev: 0x0
-- next: 0x733e25a020

s_sockethead: 0x733e25a020
-- socketfd: 0x7
-- prev: 0x733e25a000
-- next: 0x733e25a040

s_sockethead: 0x733e25a040
-- socketfd: 0x4b190e3a00000009
-- prev: 0x733e25a020
-- next: 0x0
```

Figure 13. Frida output showing where the data structure is getting corrupted

```
s_sockethead: 0x733e25a000
-- socketfd: 0x5
-- prev: 0x0
-- next: 0x733e25a040
s_sockethead: 0x733e25a040
-- socketfd: 0x5a26a02512bf1e53
-- prev: 0xea47398a5e0d32ac
-- next: 0xe8fe60173b26033a
s_sockethead: 0xe8fe60173b26033a
Error: access violation accessing 0xe8fe60173b26033a
```

Figure 14. Frida output showing a very corrupted data structure

Ultimately this led to our access violation crash. All this instrumentation and debugging data was done on the target without ever having to restart the service. Because of the nature of how this application was getting started, it did not lend itself well to being started manually, and since it was running on an Android, the `LD_PRELOAD` method of hooking would have been a major pain. Frida was able to connect to the running daemon without interrupting anything and not triggering any watchdogs. With the instrumentation Frida provided above it was easy to identify and determine the exploitability of the crash in question.

Concluding thoughts:

Hooking functions to bypass, modify, or simply gather information is a great tool that every security researcher should have in their arsenal. While there are many other useful reasons to hook functions and many other tools to do so, we hope that this can give some inspiration on what the technique of hooking can do for you in the future.

To continue learning more about Frida checkout these resources.

Frida-boot

Frida Quick Start Guide

Getting Started with Frida : Hooking a Function and Replacing its Arguments