Detecting and Bypassing Frida Dynamic Function Call Tracing: Exploitation and Mitigation[*]

Enrique Soriano-Salvador,Gorka Guardiola-Múzquiz
Universidad Rey Juan Carlos
`enrique.soriano@urjc.es, gorka.guardiola@urjc.es`

Abstract

Frida is a powerful dynamic analysis tool that uses different mechanisms to hijack the control flow of the analyzed process and is capable of communicating with external tools. The code of the process is manipulated to intercept the function calls and analyze them. Frida is

[*]This work is partially funded under the Proyectos de Generación de Conocimiento 2021 call of Ministry of Science and Innovation of Spain co-funded by the European Union, project PID2021-126592OB-C22 CASCAR/DMARCE.

commonly used to analyze suspicious programs and malware. Nevertheless, the function call interception mechanisms can be circumvented by malicious code. In this paper, we describe the different techniques to detect Frida and a novel technique to bypass those interception mechanisms. We also describe a generic mitigation method based on standard Linux capabilities, specifically the Page Table Entry (PTE) inspection mechanisms. This method is generic and does not depend on specialized hardware. Finally, we present an open source implementation, `gopper`, a lightweight stand-alone tool that watches a process to detect anomalous and suspicious behaviors without interference.

Keywords: Cybersecurity, antianalysis, antifrida, malware, evasion.

1 Introduction

Frida [1, 2, 3] is a dynamic instrumentation toolkit for reverse engineering and security analysis. It uses different mechanisms and can operate in three modes (embedded, injected and preloaded). For example, the default mode is the *injected* one. This mode uses `ptrace` to hijack the control flow of the analyzed process. Then, it allocates some memory in the process memory space and injects a library (a shared object): the *agent*. This agent uses interprocess communication (IPC) mechanisms (e.g. named pipes and signals) to communicate with external tools. The code of the process is manipulated to intercept the function calls and analyze them (using mechanisms like `mmap`, `dlopen` and `dlsym`).

One of the tools, `frida-trace`, is used for dynamically tracing function calls. It is based on the instrumentalization of the functions' preludes: the first instructions of the traced function are modified to redirect the control flow to the components injected by Frida.

This tool is highly convenient for analyzing malware. Nevertheless, current advanced malware (*split personality malware* or *evasive malware* [4]) usually includes anti-analysis mechanisms. These malicious programs try to detect if they are being analyzed in order to change their behavior: they only perform the malicious actions (e.g. extract/decrypt and execute the malicious payload) when they are not under analysis. For example, modern malware trying to detect if the environment is virtualized (i.e. if it is executing on a virtual machine or a sandbox) by checking the size of the storage devices, some processor attributes [5], the temperature of the hardware [6], some configuration files and drivers, DMI (Desktop Management Interface) data, loaded libraries, performing side-channel time attacks [7], and so on.

In the same way, advanced evasive malware can use *antifrida* techniques to detect and avoid the analysis. In this work, we describe the different *antifrida* detection techniques. Then, we focus on a simple technique that can be used to detect and bypass the `frida-trace` interception mechanisms (for all or just some selected malicious function calls). This technique consists of checking and restoring the preludes of functions that have been instrumented by `frida-trace`. This technique can be easily implemented by the malware and does not require any special privilege: the malware code only has to modify some parts of the process' memory (i.e. change and restore the function's prelude) to bypass the interception.

Dynamically detecting prelude modifications is not trivial, because the malicious process only needs to write on its own memory. It just needs to:

1. Change the `frida-trace` prelude to bypass the function tracing.

2. Call the selected function.

3. Restore the `frida-trace` prelude.

Note that, in order to detect the modification, the prelude must be checked between steps 1-3. Polling the process' memory is not feasible: it is very expensive, because it requires extra control flows performing busy waiting. Moreover, there is an obvious race condition.

In this paper, we describe a mitigation method that watches the preludes. It is based on standard Linux mechanisms and it does not depend on specialized hardware or third party tools. The method can be included in existing analysis tools (e.g. Frida) and antimalware systems. Specifically, we propose to monitor a Page Table Entry (PTE) attribute, the *soft dirty bit*, to detect prelude modifications.

In addition, we provide an open source implementation of the proposed method. The implementation also tracks changes in the permissions of the corresponding memory pages and monitors selected system calls. `Gopper` is a userspace tool designed to watch a process and detect suspicious behaviors while it is being analyzed with other tools. This command does not interfere with the watched process and can be used with other reversing and analysis tools (e.g. Frida) to detect evasion techniques and other malicious conducts.

In short, the contributions of this paper are:

- A compilation of known *antifrida* detection techniques.

- The description of a novel and effective *antifrida* technique for bypassing dynamic function call tracing.

- A generic countermeasure for this evasion technique, based on the inspection of the Page Table Entry's *soft dirty bit*.
- The description of a lightweight implementation for Linux and an open source userspace tool based on standard mechanisms.

The rest of the paper is organized as follows: Section 2 discusses related work, Section 3 explains the *antifrida* techniques for detection, Section 4 describes the technique for bypassing `frida-trace`'s interception, Section 5 describes the mitigation method, Section 6 explains our implementation and Section 7 presents the conclusions.

2 Related Work

Filho et al. [4] presented a complete study on evasion and countermeasures techniques to detect dynamic binary instrumentation (DBI) frameworks. They analyze previous taxonomies of evasive techniques ([8, 9, 10, 11, 12]) and present a new one. They also review a set of anti-instrumentation and evasion techniques for DBI [12, 13, 14, 10, 9, 15, 16, 11]. For an intensive review of general DBI evasion techniques and countermeasures, please refer to this work.

As far as we know, there are few academic works describing or discussing specific *antifrida* techniques. Druffel et al. [17] explains that interception can be bypassed with inline assembly system calls. They present DaVinci, an Android kernel space tool for dynamic application analysis. On the other hand, we describe a userspace solution to detect `frida-trace`'s interception evasion. Userspace tools reduce complexity (specially in a monolithic kernel like Linux). They are more convenient for system stability (whole system crashes, side effects, etc.), process isolation and protection, maintainability, portability, and usability in general (testing, scripting, etc.).

The OWASP Mobile Security Testing Guide [18] describes different *antifrida* techniques. Section Testing Reverse Engineering Tools Detection (MSTG - RESILIENCE - 4) explains several techniques for Android and iOS systems. Mueller provides more details [19] and source code for some of those methods [20]. Other blog posts [21, 22, 23] also describe the same detection techniques and others. All those techniques are explained in Section 3.

The OWASP document mentions the possibility of detecting trampolines and named pipes to detect Frida, but it does not provide further details. Others [21, 22, 23] also sketch this detection technique. In fact, detecting trampolines in preludes is a well known method to detect *hooking* in general. For example, it can be used to prevent bypassing SSL certificate pinning [24]. Nevertheless, none of these publications describes techniques to bypass `frida-trace` dynamic interception We describe a technique to evade the interception, based on changing and restoring the `frida-trace`'s trampolines. We also provide a countermeasure.

Other systems propose the use of the PTE attributes for memory checkpointing [25, 26]. For example, CRIU [26] (used by OpenVZ, LXC/LXD and Docker) permits freezing a running container (or an application), checkpointing its state to disk, and restoring it later. It uses the *soft dirty bit* to track memory changes in this process. We propose the same tracking mechanism as a countermeasure to `frida-trace` interception bypassing.

3 Antifrida: detection techniques

In this section, we briefly describe the known *antifrida* detection techniques.

3.1 Loaded libraries

Inspecting the loaded libraries of a process to find Frida libraries is a straightforward detection method. As described in [18], the malware can inspect the memory space of the process and find the regions for the mapped library (the *agent*).

For example, if we run `frida-trace` to intercept the calls for `open` (the `libc`'s stub for the `open` system call) and dump the memory map of the corresponding process, we can see the Frida library:

```
$> frida-trace -i open a.out &
...
$> cat /proc/12555/maps | grep frida
7fea7168a000-7fea72c53000 r-xp 00000000 103:04 1968850
    /tmp/frida-81be93df197f7fb45dea328c7237c270/frida-agent-64.so
7fea72c53000-7fea72c54000 ---p 015c9000 103:04 1968850
    /tmp/frida-81be93df197f7fb45dea328c7237c270/frida-agent-64.so
7fea72c54000-7fea72cde000 r--p 015c9000 103:04 1968850
    /tmp/frida-81be93df197f7fb45dea328c7237c270/frida-agent-64.so
7fea72cde000-7fea72cf6000 rw-p 01653000 103:04 1968850
    /tmp/frida-81be93df197f7fb45dea328c7237c270/frida-agent-64.so
$>
```

The malware could inspect its own memory maps to do the same. Nevertheless, it would require some suspicious system calls that would alert the analyst (open the `/proc` file, reading it, etc.).

3.2 Package signatures

Checking the signatures of the Android package is another technique described by the OWASP document [18]. To include the instrumentation mechanisms in the software, the application package has to be rebuilt.

If it was signed, the signature (which will change with any modification of the package data) can be checked to detect that the application has been modified by Frida.

3.3 Frida resources

Checking the environment for related artifacts is also a well known detection technique [18, 19, 20, 21].

The malware can inspect the system and find Frida's resources, such as processes, files (e.g. `.so` libraries), temporal files, services (e.g. `frida-server`) and so on.

The malware can also inspect the network resources to detect Frida. The common technique is checking if the TCP 27042 port is open.

Frida also uses the D-Bus protocol to communicate. Checking for ports responding to D-Bus Auth is also effective to detect Frida.

The components `frida-gadget` and `frida-server` create specific named threads (e.g. `gmain`, `gum-js-loo`) [21, 23]. The threads can be detected by inspecting:

```
/proc/<pid>/task/<tid>/status
```

In addition, Frida also uses named pipes (i.e. fifos) [21]. The malware can read the directory:

```
/proc/<pid>/fd
```

This way, it can find those pipes and other file descriptors related to Frida. For example, Frida uses temporal files located in:

```
/data/local/tmp/
```

Another common antidebugging technique is checking the parent process. Commonly, the parent process is the shell, a desktop session or `init` if it inherits it after the parent exits (`bash`, `sh`, `gnome-session`, `systemd`, in Linux or `explorer.exe` in Windows). The malware can check if a Frida tool is the program executed by the parent process [22].

3.4 Memory artifacts and function preludes

Scanning a process memory for artifacts, such as strings (e.g. "LIBFRIDA"), is also a known technique [18], as is checking exported functions' names [22].

`Frida-trace`'s trampolines are memory artifacts too. The beginning of the function is replaced by a trampoline code. Once the call is intercepted, the flow is redirected to the Frida agent. Then, the agent can execute whatever it needs: incrementally, for each branch in the function, a component named *stalker* rewrites the code basic blocks and stores them in a new executable page. This way, it can add new code in different parts of the intercepted function.

Comparing the code section in memory with the code section of the library files (standard C library and native library) [21] is is a (heavy) way to check changes in the preludes.

There are more subtle ways to do the same without performing suspicious system calls (i.e. to read the `/proc` files, the excutable file or the libraries).

Suppose the following example[1]:

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

void
dumpprelude(unsigned char *f)
{    int i;
    int c;

    for(i=0; i<32; i++){
        fprintf(stderr, "%02x", *(f+i));
    }
    fprintf(stderr, "\npress enter...\n");
    read(0, &c, 1);
}

int
f(int x)
{
    fprintf(stderr, "f says hi!\n");
    return ++x;
}

int
main(int argc, char *argv[])
{
    int x;

    dumpprelude((unsigned char*)f);
    dumpprelude((unsigned char*)f);
    x = f(13);
    fprintf(stderr, "x is %d\n", x);
    exit(EXIT_SUCCESS);
}
```

[1]Be careful when writing this kind of code: it is very easy to hit undefined behaviour.

If we compile and execute the program, pressing enter when prompted:

```
$> gcc -o ex1 ex1.c
$> ./ex1
f30f1efa554889e54883ec10897dfc488b05942d00004889c1ba0b000000be01
press enter...

f30f1efa554889e54883ec10897dfc488b05942d00004889c1ba0b000000be01
press enter...

f says hi!
x is 14
$>
```

In this case, we can see that the prelude of the function does not change. If we disassemble the function `f`, we can see the original instructions:

```
0x00001276      f30f1efa       endbr64
0x0000127a      55             pushq %rbp
0x0000127b      4889e5         movq %rsp, %rbp
0x0000127e      4883ec10       subq $0x10, %rsp
0x00001282      897dfc         movl %edi, -4(%rbp)
```

If we execute it again:

```
$> ./ex1
f30f1efa554889e54883ec10897dfc488b05942d00004889c1ba0b000000be01
press enter...
```

Before pressing enter, we run `frida-trace` as root in another terminal to trace the function `f` (offset 0x1276 in the binary `text` segment):

```
$> nm ex1 | grep ' f$'
0000000000001276 T f
$> frida-trace -a ex1\!0x1276 ex1
Instrumenting...
sub_1276: Auto-generated handler at
    "/tmp/__handlers__/ex1/sub_1276.js"
Started tracing 1 function. Press Ctrl+C to stop.
```

Then, we press enter in the other terminal:

```
e98d4d00004889e54883ec10897dfc488b05942d00004889c1ba0b000000be01
press intro...

f says hi!
x is 14
$>
```

We can observe that the prelude has changed: the first 5 bytes were `f30f1efa55`, now they are `e98d4d0000`. The first two instructions of function `f` have been replaced to jump to the Frida code. In this example, Frida will intercept the call to `f`:

```
...
frida-trace intercepts one call (f is named sub_1276()):
           /* TID 0x38f4 */
  7393 ms  sub_1276()
Process terminated
$>
```

This technique does not require any external function or system call. The malware only needs to check some bytes of its own code. If any of the preludes has been changed, then `frida-trace` is present in the system.

4 Evasion

If the malware detects Frida, it can try to bypass the function call interception. The maliciuos binary can restore the original prelude before calling the function. Moreover, the malware can hide just the important calls (those that perform the suspicious malicious actions) and let Frida intercept the rest of calls to deceive the analyst.

If we modify the previous example to restore the original prelude:

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

void
dumpprelude(unsigned char *f)
{
    int i;
    int c;

    for(i=0; i<32; i++){
        fprintf(stderr, "%02x", *(f+i));
    }
    fprintf(stderr, "\npress enter...\n");
    read(0, &c, 1);
}

int
f(int x)
{
    fprintf(stderr, "f says hi!\n");
    return ++x;
}

int
main(int argc, char *argv[])
{
    int x;

    dumpprelude((unsigned char*)f);
    dumpprelude((unsigned char*)f);
    memcpy((void*)f, "\xf3\x0f\x1e\xfa\x55", 5);
    dumpprelude((unsigned char*)f);
    x = f(13);
    fprintf(stderr, "x is %d\n", x);
    exit(EXIT_SUCCESS);
}
```

Then, we execute it (running `frida-trace` in another terminal as in the previos example)[2]:

```
$> ./ex1
f30f1efa554889e54883ec10897dfc488b05742d00004889c1ba0b000000be01
press enter...

e96d4d00004889e54883ec10897dfc488b05742d00004889c1ba0b000000be01
press enter...

f30f1efa554889e54883ec10897dfc488b05742d00004889c1ba0b000000be01
press enter...

f says hi!
x is 14
$>
```

Now, `frida-trace` does not intercept the call:

```
$> frida-trace -a ex1\!0x1296 ex1
Instrumenting...
sub_1296: Auto-generated handler at
      "/tmp/__handlers__/ex1/sub_1296.js"
Started tracing 1 function. Press Ctrl+C to stop.
Process terminated
$>
```

Note that the malware does not need to use the `memcpy` function to restore the prelude, it could use a simple loop like[3]:

```
for(i=0; i<len; i++, dest++, src++){
    *dest = *src;
}
```

This way, it would not depend on any library function to restore the prelude.

The attentive reader will note that the program has modified a `text` executable memory page (i.e. instructions) and there was not an execution error. By default, Linux and other modern operating systems disable the write permission for executable memory pages. This way, if the process tries to write the code, it receives a `SIGSEGV` signal and crashes (*segmentation*

[2]Note that the offset has changed: now it is 0x1296.

[3]Being `len` the number of bytes to be copied, `dest` a `char` pointer with the destination address, and `src` a `char` pointer with the source address.

fault). That did not happen. Why? Because Frida needs to change the permissions of the `text` pages to instrument the code. After modifying the functions' preludes, Frida leaves the write permission enabled. If we check the memory map, the code region has write, read and execution permissions (`rwxp`):

```
56550bc57000-56550bc58000 rwxp 00001000 103:02 4724567  /tmp/ex1
```

That is very convenient for the evasion. Nevertheless, it is not a requirement to implement the attack. The malware would use the `mprotect` system call to enable the write permission to rewrite the preludes and bypass interception. Note that using `mprotect` to enable the write permission for an executable page would be a striking clue of malicious activity (but the attack is viable).

As far as we know, no previous malware has implemented this *antifrida* evasion technique yet.

4.1 Proof of Concept

We present a proof of concept based on a simple C macro, which can be used by the malware to perform silent calls to selected functions. It has the following parameters:

- Variable to store the return value.
- Function to be called.
- Array with the original prelude for the function.
- Arguments for the function (variadic).

```
#define HIDECALL(RET,FUNC,PRELUDE,...) \
    {   int i; \
        int presz = sizeof(PRELUDE); \
        unsigned char aux[presz]; \
         for(i = 0; i < presz; i++) {\
            if(*(((unsigned char*)FUNC)+i) != PRELUDE[i]){\
                break;\
            } \
        } \
        if(i != presz){ \
            for(i = 0; i < presz; i++){ \
                aux[i] = *(((unsigned char*)FUNC)+i); \
                *(((unsigned char*)FUNC)+i) = PRELUDE[i]; \
            } \
            RET = FUNC(__VA_ARGS__); \
            for(i = 0; i < presz; i++){ \
                *(((unsigned char *)FUNC)+i) = aux[i]; \
            } \
        }else{ \
            RET = FUNC(__VA_ARGS__); \
        } \
    }
```

This macro:

1. Checks if the prelude is the original one
2. If not, it copies the original one, calls the function and restores the Frida prelude
3. Else, it calls the function normally (Frida is not present in the system)

A malicious binary could use the macro to hide the important calls (i.e. evidences of malicious activity).

4.2 Example of use

The following program dumps a file (its path is passed as an argument). If the `-s` option is used, it dumps the file silently: all the calls to the `libc` functions `open`, `read`, `write` and `close` are hidden (i.e. they will not be intercepted). These functions will perform the required system calls to read and write the file.

If the `-s` option is not used, the `libc` functions are called normally.

The original preludes for `open`, `read`, `write` and and `close` are stored in global variables:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <err.h>
#include <unistd.h>
#include <fcntl.h>
#include "hidecall.h"


enum{
    Bsz = 1024,
};

unsigned char openprelude[] = {
                0xf3, 0x0f, 0x1e, 0xfa, 0x41, 0x54,
                0x41, 0x89,
                };

unsigned char readprelude[] = {
                0xf3, 0x0f, 0x1e, 0xfa, 0x64, 0x8b,
                0x04, 0x25, 0x18, 0x00, 0x00, 0x00,
                };
```

```
unsigned char writeprelude[] = {
                0xf3, 0x0f, 0x1e, 0xfa, 0x64, 0x8b,
                0x04, 0x25, 0x18, 0x00, 0x00, 0x00,
                };

unsigned char closeprelude[] = {
                0xf3, 0x0f, 0x1e, 0xfa, 0x64, 0x8b,
                0x04, 0x25, 0x18, 0x00, 0x00, 0x00,
                };

void
usage(void)
{
    fprintf(stderr, "usage: readfile [-s] path\n");
    exit(EXIT_FAILURE);
}

void
readall(char *path)
{
    int fd;
    int nr;
    char buf[Bsz];

    fd = open(path, O_RDONLY);
    if(fd < 0){
        err(EXIT_FAILURE, "open failed");
    }
    while((nr = read(fd, buf, Bsz)) > 0){
        if(write(1, buf, nr) != nr){
            err(EXIT_FAILURE, "write failed");
        }
    }
    if(nr < 0){
        err(EXIT_FAILURE, "read failed");
    }
    close(fd);
}

void
silentreadall(char *path)
{
    int fd;
    int nr;
    char buf[Bsz];
    int ret;

    HIDECALL(ret, open, openprelude, path, O_RDONLY);
    fd = ret;
    if(fd < 0){
        err(EXIT_FAILURE, "open failed");
    }
    for(;;){
        HIDECALL(ret, read, readprelude, fd, buf, Bsz);
        nr = ret;
        if(nr <= 0)
            break;
        HIDECALL(ret, write, writeprelude, 1, buf, nr);
        if(ret != nr){
            err(EXIT_FAILURE, "write failed");
        }
    };
    if(nr < 0){
        err(EXIT_FAILURE, "read failed");
    }
    HIDECALL(ret, close, closeprelude, fd);
}

int
main(int argc, char *argv[])
{
    char c;

    fprintf(stderr, "press enter...\n");
    read(0, &c, 1);

    argc--;
    argv++;
    if(argc < 1 || argc > 2){
        usage();
    }
    if(argc == 2){
        if(strcmp(argv[0], "-s") != 0){
            usage();
        }
        silentreadall(argv[1]);
    }else{
        readall(argv[0]);
    }
    exit(EXIT_SUCCESS);
 }
```

If we run it without the `-s` option and execute `frida-trace` before pressing enter:

```
$> frida-trace -i open -i read -i write -i close readfile-simple
Instrumenting...
open: Loaded handler at "/tmp/__handlers__/libc_2.31.so/open.js"
read: Loaded handler at "/tmp/__handlers__/libc_2.31.so/read.js"
write: Loaded handler at "/tmp/__handlers__/libc_2.31.so/write.js"
close: Loaded handler at "/tmp/__handlers__/libc_2.31.so/close.js"
open: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/open.js"
read: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/read.js"
write: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/write.js"
close: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/close.js"
Started tracing 8 functions. Press Ctrl+C to stop.
           /* TID 0x4445 */
  1986 ms  open(pathname="/tmp/f", flags=0x0)
  1986 ms  read(fd=0x4, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  write(fd=0x1, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  read(fd=0x4, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  write(fd=0x1, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  read(fd=0x4, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  write(fd=0x1, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  read(fd=0x4, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  write(fd=0x1, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  read(fd=0x4, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  write(fd=0x1, buf=0x7ffcb03292f0, count=0x388)
  1986 ms  read(fd=0x4, buf=0x7ffcb03292f0, count=0x400)
  1986 ms  close(fd=0x4)
Process terminated
$>
```

In this case, `frida-trace` is able to intercept the calls. On the other hand, if we execute the program with the `-s` option:

```
$> frida-trace -i open -i read -i write -i close readfile-simple
Instrumenting...
open: Loaded handler at "/tmp/__handlers__/libc_2.31.so/open.js"
read: Loaded handler at "/tmp/__handlers__/libc_2.31.so/read.js"
write: Loaded handler at "/tmp/__handlers__/libc_2.31.so/write.js"
close: Loaded handler at "/tmp/__handlers__/libc_2.31.so/close.js"
open: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/open.js"
read: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/read.js"
write: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/write.js"
close: Loaded handler at "/tmp/__handlers__/libpthread_2.31.so/close.js"
Started tracing 8 functions. Press Ctrl+C to stop.
Process terminated
$>
```

Now, the hidden calls have not been intercepted and the program works correctly:

```
$> seq -w 1 1000 > /tmp/f
$> ./readfile-simple -s /tmp/f
press enter...
0001
0002
...
0996
0997
0998
0999
1000
$>
```

5 Mitigation

Let's generalize the problem: a process P executes a program B that changes a variable var. Later, it restores the previous value:

```
aux=var;
var=z;
dosomething();
var=aux;
```

How could we track memory changes to detect this modification of variable var?

Polling P's memory is not feasible. We would need a separate control flow P' reading var in a loop to detect modifications (i.e. busy waiting). It does not work because there is a race condition: the main flow can change and restore var between two consecutive checks. If the polling frequency is low, the probability of failing to detect the change is very high. If the frequency is high, it is very expensive (and the race condition still exists). Synchronization (e.g. using a mutex) is not possible, because we do not control B (the malware does in our specific case).

Modern processors provide instructions to monitor memory addresses, for example `umwait` and `umonitor` [27]. This approach is not viable. These instructions require changes in the state of the core and are very expensive: A `umwait` instruction tells the processor to stop executing until the corresponding write occurs. This mechanism is prohibitive.

We have another option: consult the attributes of the corresponding memory page. Modern processors use memory paging. Pages have different attributes for memory management. Among them, there is the dirty bit, which is set to 1 when the page is written. This bit is used to implement page replacement algorithms. We can use it to detect changes in a page.

For the general problem, we should locate the variable *var* in a dedicated memory page. Once *var* is initialized, the dirty bit of this page should be cleared. Then, we can poll the dirty bit to detect changes in *var*.

In our specific case (i.e. detecting function modifications), we don't need a dedicated page for each function, because all the executable pages of the process should be read-only.

The method is simple:

1. After instrumenting the preludes, the write permission is disabled from all executable pages (`text` segment, libraries, etc.).
2. The dirty bit is cleared for all these pages.
3. Concurrently:
   (a) Monitor the dirty bit of the corresponding pages (the ones containing the functions).
   (b) Monitor attempts to clear the dirty bit.
   (c) Monitor changes in the permissions of these pages.

6 Implementation for Linux

In particular, AMD64 uses a multilevel page table to translate virtual addresses. In the last level, a Page Table Entry (PTE) points to a physical memory page (if 4 KB pages are used). The PTE includes several bits (present, execution, writable, dirty, etc.).

Linux manages the *soft dirty bit* [28]. The soft dirty bit behaves like the dirty bit, but it requires kernel support. The kernel clears the writable bit

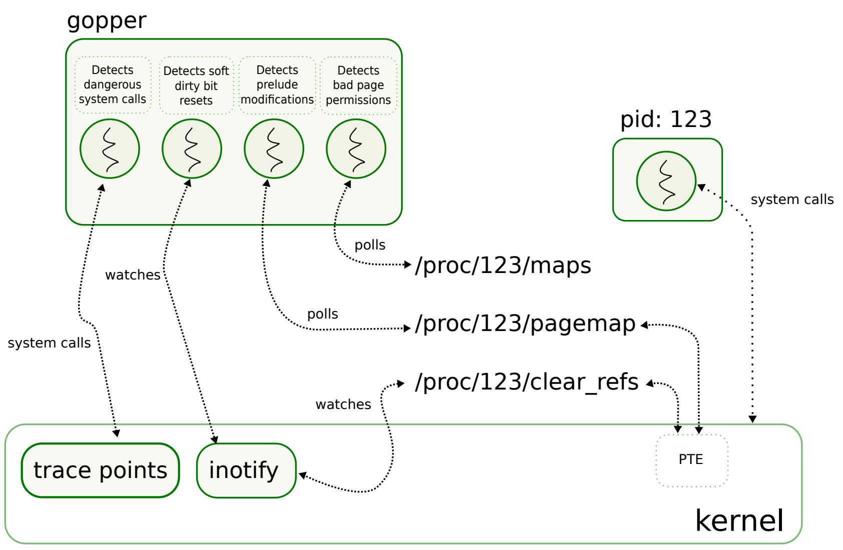

Figure 1: `gopper` watching a process (PID: 123). It receives events from `inotify` and `trace-points` and polls the `/proc` files to detect dangerous permissions for pages and modifications of the watched memory addresses (i.e. Frida preludes).

from the PTE when the soft dirty bit is cleared. Later, when the process tries to write the page, a page fault occurs. Then, the kernel handles the page fault, sets the soft dirty bit in the PTE and resumes the process execution.

At first glance, the obvious choice would be an implementation within the Linux kernel (to read and write de PTE data). A kernel module could implement the method described above (or provide an interface for a userspace program). Fortunately, there is an alternative for implementing it in userspace.

Since Linux 3.11, the soft dirty bit can be retrieved and cleared from userspace. It is done through the `/proc` filesystem. To get the soft dirty bit from userspace, we can read:

`/proc/<pid>/pagemap`

The bit 55 of this 64-bit word is the soft dirty bit. According to the interface defined in the `/proc` manua;[4], to clear the bit we must write the text value "4" in:

`/proc/<pid>/clear_refs`

Therefore, the method can be fully implemented by a userspace program. We have solved only part of the problem. There are two issues:

[4]man 5 proc

1. The granularity is 4 KB (page size).
2. The malicious program can clean the bit after modifying and restoring the preludes.

In the `frida-trace` case, issue (1) is not a problem. Code pages (e.g. `text` segment pages, libraries, etc.) should not be writable (although `frida-trace` leaves the write permission enabled, as explained previously). Code pages should not be modified after Frida's instrumentation. Any further modification can be evidence of malicious behavior.

Issue (2) can be solved by watching accesses to the `clear_refs` file. Fortunately, the `/proc` filesystem supports the `inotify` API [29].

6.1 Gopper

As stated before, we implemented a new userspace tool named `gopper` to detect the *antifrida* evasion technique described in Section 4 by implementing the mechanisms described in Section 5. The tool is written in Go[5] and it is libre (free) software.

This program must be run with root privileges. It spawns different *goroutines* (i.e. concurrent control flows) to:

- Check modifications of the corresponding memory pages. This is done by polling the pages' soft dirty bit, available through `/proc/<pid>/pagemap` as explained before.
- Monitor operations over `/proc/<pid>/clear_refs`. The malicious program could use this file to hide some page changes (by changing a page and clear the soft dirty bit while the polling is done, i.e. between two reads).

  This is done by receiving the `inotify` events. Gopper uses the Kubernetes[6] Go package for the `inotify` mechanisms.
- Detect modifications of the permissions of the pages. It also warns about dangerous permissions (i.e. write and exec permissions).

  To do that, it polls the `/proc/<pid>/maps` file.
- Trace the `mprotect` system call, used to change page permissions in Linux, and other system calls defined by the user.

[5]https://go.dev
[6]`k8s.io/utils/inotify`

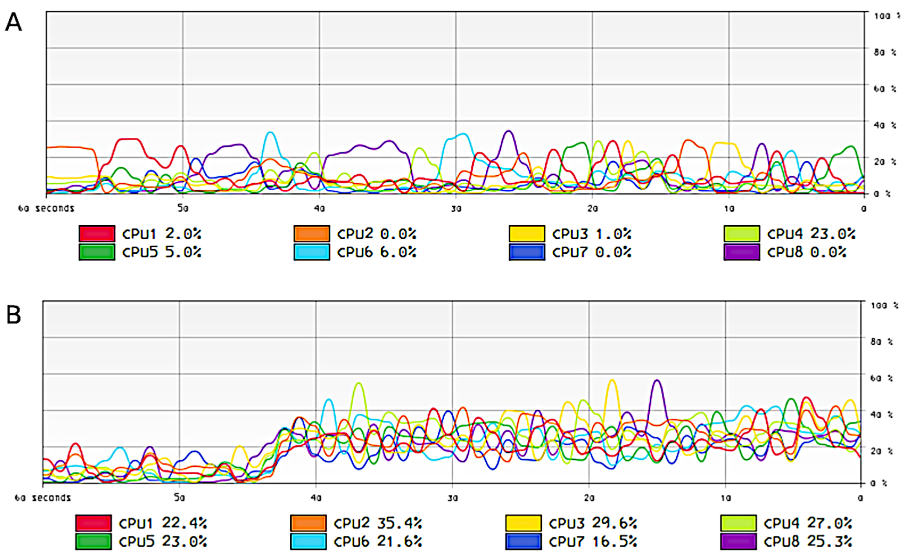

Figure 2: (A) `gopper` watching a process reading a file with 100,000 lines (starting at x-axis value 40 and finishing at value 33). (B) `gopper` watching a process reading a file with infinite lines (starting at x-axis value 44).

This is done by receiving events from the Linux kernel tracepoints [30] through the synthetic files located in `/sys/kernel/debug/tracing`.

The tool does not interfere with the watched process (it doesn't inject artifacts in the process memory, etc.). All these actions are performed through external standard Linux mechanisms. Figure 1 depicts an example. The command be used together with Frida-trace or any other analysis tool.

`Gopper` is lightweight, because:

1. The polling frequency can be low. No matter what polling time we set, prelude modifications will still be detected (sooner or later) because the attacker is not able to clear the dirty bit silently (`gopper` will receive this event). By default, polling time is set to 400 ms.

2. `inotify` and the Linux kernel tracepoints provide events, so these mechanisms do not add noticeable overhead.

The command requires two mandatory arguments: the process identifier (PID) and at least one memory address to be watched (i.e. the memory addresses of the functions we want to watch, that can be found with any standard debugger). It admits other optional arguments, such as the polling frequency and a list of extra system calls to be monitored.

In the following example, PID 24841 runs the program explained in Section 4.2, which is being analyzed by `frida-trace`. It is executed with the `-s` flag, so the program modifies the Frida preludes in order to bypass the interception mechanisms. In addition, the soft dirty bit is cleared of the corresponding pages are cleared by another process:

```
$: gopper -addr=7f777e5c61d0,7fffe69c6757 -syscalls=clone,read 24841
2021/12/15 20:13:30 watching pid: 24841
2021/12/15 20:13:30 watching address: 7f777e5c61d0
2021/12/15 20:13:30 watching address: 7fffe69c6757
2021/12/15 20:13:30 critical: 7f777e5c61d0 page writable and executable
2021/12/15 20:13:34 warning: mprotect detected for 7fffe69c6757
2021/12/15 20:13:34 critical: mprotect w+x detected for 7fffe69c6757
2021/12/15 20:13:34 warning: syscall read detected:
   readfile-24841   [001] .... 15133.617503:
        sys_read(fd: 4, buf: 7fffe69c6330, count: 400)
2021/12/15 20:13:34 warning: syscall read detected:
   readfile-24841   [001] .... 15133.617559:
        sys_read(fd: 4, buf: 7fffe69c6330, count: 400)
2021/12/15 20:13:34 warning: syscall read detected:
   readfile-24841   [001] .... 15133.618244:
sys_read(fd: 0, buf: 7fffe69c6787, count: 1)
2021/12/15 20:13:34 critical: page of address 7f777e5c61d0
                   was modified
2021/12/15 20:13:34 critical: page of address 7fffe69c6757
                   was modified
2021/12/15 20:14:26 critical: soft dirty could be cleared
                  "/proc/24841/clear_refs": 0x2 == in_modify
2021/12/15 20:14:26 critical: soft dirty could be cleared
                  "/proc/24841/clear_refs": 0x20 == in_open
2021/12/15 20:14:26 critical: soft dirty could be cleared
                  "/proc/24841/clear_refs": 0x2 == in_modify
2021/12/15 20:14:26 critical: soft dirty could be cleared
                  "/proc/24841/clear_refs": 0x8 == in_close_write
2021/12/15 20:14:32 process (24841) died
$>
```

To convey an idea of the CPU usage, Figure 2(A) shows the 8 CPUs of a Intel Core i5-10210U CPU at 1.60GHz laptop with 8GB of RAM executing `gopper` as in the last example (i.e. watching two preludes and monitoring two system calls, `clone` and `read`). In this case, the watched process just reads a plain text file with 100,000 lines (writing them to the console). The execution starts at the x-axis value 40 and finishes at the x-axis value 33. As shown in the figure, the impact is negligible. Figure 2(B) shows the execution of the same program, but reading an infinite number of lines. The program starts at the x-axis value 44. We can observe that, in this extreme case, the system is not overloaded (none of the CPUs are over the 60% of usage).

As shown, using `gopper`, the analyst is able to detect that the malicious program is bypassing the Frida interception mechanisms by changing and restoring the functions' preludes. Note that `gopper` can also be used to detect other *antifrida* techniques described in section 3, for example, monitoring operations over `/proc` files (to find the Frida file descriptors, threads and so on).

7 Conclusions

In this paper we describe known *antifrida* detection techniques and a simple but effective technique to bypass dynamic function call tracing without any special privilege, function or system call.

We also provide the description of a generic method to detect this *antifrida* evasion technique. The method is based on standard OS mechanisms (i.e. the Page Table Entry attributes), it does not depend on specialized hardware and can be implemented by existing analysis/reverse engineering tools and antimalware solutions.

Last, we present a functional implementation of the proposed method: a simple, stand-alone Linux command named `gopper`. This implementation is lightweight and runs in userspace. It is written in Go and it is libre software. Its source code can be downloaded from:

`https://gitlab.etsit.urjc.es/esoriano/gopper`

Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

References

[1] K. T. Kalleberg, “Frida: Putting the open back into closed software,” 2015, open Source Developers Conference, OSDC Nordic 2015.

[2] A. O. A. V. Ravnas, “The engineering behind the gnireenigne,” 2015, open Source Developers Conference, OSDC Nordic 2015.

[3] “Frida github,” 2022, https://github.com/frida/frida.

[4] A. S. Filho, R. J. Rodríguez, and E. L. Feitosa, “Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks,” *Digital Threats*, vol. 3, no. 2, feb 2022. [Online]. Available: https://doi.org/10.1145/3480463

[5] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st ed. USA: No Starch Press, 2012.

[6] "Gravityrat, mitre attack," https://attack.mitre.org/software/S0237/.

[7] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: Vmm detection myths and realities," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07. USA: USENIX Association, 2007.

[8] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed)," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 15–27.

[9] R. J. Rodríguez, I. R. Gaston, and J. Alonso, "Towards the detection of isolation-aware malware," *IEEE Latin America Transactions*, vol. 14, no. 2, pp. 1024–1036, 2016.

[10] K. Sun, X. Li, and Y. Ou, "Break out of the truman show: Active detection and escape of dynamic binary instrumentation," *Black Hat Asia*, 2016.

[11] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, "Pwin–pwning intel pin: Why dbi is unsuitable for security applications," in *European Symposium on Research in Computer Security.* Springer, 2018, pp. 363–382.

[12] Z. Zhechev, "Security evaluation of dynamic binary instrumentation engines," Ph.D. dissertation, Technical University of Munich Munich, Bavaria, 2018.

[13] R. J. Rodríguez, E. L. Feitosa *et al.*, "Reducing the attack surface of dynamic binary instrumentation frameworks," in *Developments and Advances in Defense and Security.* Springer, 2020, pp. 3–13.

[14] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro, "On the dissection of evasive malware," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2750–2765, 2020.

[15] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero, "Measuring and defeating anti-instrumentation-equipped malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2017, pp. 73–96.

[16] M. Hron and J. Jermář, "Safemachine: Malware needs love, too," *Virus Bulletin*, 2014.

[17] A. Druffel and K. Heid, "Davinci: Android app analysis beyond frida via dynamic system call instrumentation," in *Applied Cryptography and Network Security Workshops*, J. Zhou, M. Conti, C. M. Ahmed, M. H. Au, L. Batina, Z. Li, J. Lin, E. Losiouk, B. Luo, S. Majumdar, W. Meng, M. Ochoa, S. Picek, G. Portokalidis, C. Wang, and K. Zhang, Eds. Cham: Springer International Publishing, 2020, pp. 473–489.

[18] "Mobile security testing guide," https://owasp.org/www-project-mobile-security-testing-guide/.

[19] B. Mueller, "The jiu-jitsu of detecting frida," https://web.archive.org/web/20181227120751/http://www.vantagepoint.sg/blog/90-the-jiu-jitsu-of-detecting-frida.

[20] ——, "Frida detection examples github," https://github.com/muellerberndt/frida-detection.

[21] G. Arvind, "Detect frida for android," https://darvincitech.wordpress.com/2019/12/23/detect-frida-for-android/.

[22] NCR, "Anti-instrumentation techniques: I know you're there, frida!" https://crackinglandia.wordpress.com/2015/11/10/anti-instrumentation-techniques-i-know-youre-there-frida/.

[23] R. Thomas, "r2-pay: anti-debug, anti-root and anti-frida," https://www.romainthomas.fr/post/20-09-r2con-obfuscated-whitebox-part1/.

[24] D. Frett, "Prevent bypassing of ssl certificate pinning in ios applications," https://www.guardsquare.com/blog/iOS-SSL-certificate-pinning-bypassing.

[25] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Lightweight memory checkpointing," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 474–484.

[26] "Criu," https://criu.org/Main_Page.

[27] "Intel intrinsics guide," https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=UMWAIT.

[28] "The linux kernel user's and administrator's guide," https://www.kernel.org/doc/html/v5.0/admin-guide/mm/soft-dirty.html.

[29] “inotify(7) - linux manual page.”

[30] T. Ts’o, “Event tracing,” https://www.kernel.org/doc/Documentation/trace/events.txt.