Thesis Dissertation

CLASSPIN:CLASS POINTER HIJACKING PROTECTION FOR OBJECTIVE-C BINARIES

Marios Karapetris

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2018

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

ClassPin:Class Pointer Hijacking Protection for Objective-C Binaries

Marios Karapetris

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfillment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2018

Acknowledgments

I would like to take the opportunity to express my appreciation to Dr. Elias Athanasopoulos, my thesis dissertation supervisor, for his pricelesses advices and guidance during this project. His continuous help and the excellent communication between us were critical to overcome all the challenges and difficulties I faced from the start of this dissertation until the end. My sincere thanks for giving me the opportunity to deal with Computer Security field and sharing his educational views on several issues related to my thesis or not.

In addition, I would like to express my thanks to the other members of Security Research Group of the University of Cyprus for their help during the course of the dissertation.

Finally, I have to thanks my family and my friends for their incessant support,their encouragement and for being so motivating all these years, which undoubtedly played an enormous role to complete my studies and achieve my goals in general.

Summary

Software exploitation's modern techniques usually contain a step where attackers abuse use-after-free vulnerabilities. In Objective-C programs, such attacks can be achieved by hijacking the class pointer in object's instances. In addition, class pointer hijacking bypasses the established defenses and does not involve memory corruption, as the attacker just needs to utilize dangling pointers that are still available by a process. As a result, the attackers manage to abuse the program's control flow according to their needs.

In this thesis dissertation, we analyze the above problem and we design and implement ClassPin. ClassPin is a system that provides protection against class pointer hijacking, through use-after-free vulnerabilities, to Objective-C programs. It can be used, especially, in large Objective-C binaries that are impossible to be re-compiled or re-written, as it is fully portable. The core idea of ClassPin is to *pin* every freed class pointer to a safe class under ClassPin's control so it prevents its space from future allocations and subsequently from being hijacked. Specifically, in every object destruction, ClassPin preserves and redirects the class pointer to the safe class and deallocates the rest of the space occupied by the object. Therefore, if a dangling pointer calls a method, a safe method of the safe class will be invoked. As a result, ClassPin not only defuses all dangling pointers but can also help to track and patch them.

ClassPin is suitable and practical solution to the problem, as it does not require access to the source code (*i.e.,* re-compilation) or the binary (*i.e.,*binary analysis) of the program that going to be protected. It works directly on binaries and in the least intrusive way. Also, it does not interfere with the program allocator's strategies and policies. Simply, ClassPin is invoked in each object's destruction and preserves the class pointer. As a result, ClassPin is very effective and fast. For instance, ClassPin protects Safari when running popular web browser benchmarks with an addition of 0.38% -1.93% average memory overhead and 21.12% - 28.43% performance overhead.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In days where the use of computer systems is essential for a plethora of actions in everyday life and several software applications are used from the simplest to the most vital importance procedures, it is crucial to establish that they are preserving their integrity and they are not susceptible to attacks. In the recent past, in order to secure software from being exploited, there was a great advance in software hardening which unquestionably made attacks' construction a really challenging procedure [36]. Regardless of the numerous protections in modern systems [32], such as address space layout randomization (ASLR) [34], sandboxing [24] and non-executable memory [21], attackers still manage to compromise critical applications, like web browsers.

Attackers, in order to bypass defenses in place, are taking advantage of a variety of different vulnerabilities. Abuse of temporal safety errors, especially use-after-free vulnerabilities, became very common in modern exploitation techniques [1, 2, 14, 15]. It is remarkable that hijacking the control flow of a process using use-after-free bugs does not involve corruption of memory. Instead, the attacker just has to utilize dangling pointers still accessible by a process. In addition, this type of attack neither violates the integrity of return addresses so it remains undetected by generic CFI policies [30].

Attacks based on temporal safety errors could be tremendously effective in large Objective-C programs. Objective-C is the primary programming language in Apple's OS X and iOS platforms and is used for the development of some of the main applications in Apple's platforms such as iTunes, Safari, etc. It is an extension of C language with object-oriented constructs, while differs from C++ language in the way that dynamic dispatch of function call is implemented. Each object instance holds a pointer to the class

of it, which additional contains pointers to super-class, cache and methods and they are used when the respective object calls a function.

Therefore, use-after-free attacks may be more likely in Objective-C programs than in programs implemented in other programming languages which support dynamic dispatch only in virtual objects, such as C++. For instance, assuming that there are dangling pointers in a program, an attacker could overwrite the pointer to the class, by driving the program to allocate memory with data of his choice, resulting to have the ability to redirect the control flow of the program according to his needs. Should be noticed that, abusing use-after-free bugs by hijacking objects' class can be combined with other attack vectors for delivering the end-to-end exploit.

For defending against use-after-free attacks in Objective-C, we suggest ClassPin, a protecting mechanism works in the *minimum* intrusive way. ClassPin has no need for both the source code of the program or recompilation of it, as it works directly on Objective-C binaries. Furthermore, it does not depend on complex binary analysis and does not interfere with allocator's strategies and policies, causing low overhead. Notice that, despite the several proposals for defending against these types of attacks in C++, use-after-free attacks in Objective-C has not been thoroughly examined by system security researchers.

ClassPin preserves class pointers after an object is freed and pins them to a safe object's class by instrumenting all free calls. Each time free called, ClassPin quickly detects if the memory going to be freed is associated with an object by using Objective-C Runtime API. Subsequent, the deallocation is handled by ClassPin if it is related to an object, or forwarded to the program's allocator if it is not. Through the deallocation, all the memory allocated by the object is freed except the pointer to the object's class. Also, the preserved pointer is redirected to the ClassPin's safe object. Therefore, in any trig of a dangling pointer, the safe object invokes and handles the function call. As a result, ClassPin not only provides protection against use-after-free attacks, but also helps to log and track dangling pointers.

To sum up, ClassPin handles only the deallocation of objects and only a single pointer survives (*i.e.,* 8 bytes), while all other memory operations, including allocation of objects, are handled by the program's allocator. Consequently, the memory overhead that ClassPin causes is low, as it is demonstrated in evaluation with Safari. Additionally, an idea of a garbage collector is proposed, which could periodically search for preserved pointers that are unlikely to be dangling pointers, free them and retrieve their occupied memory.

Scope. ClassPin protects Objective-C binaries from being exploited through class pointer hijacking and use-after-free vulnerabilities. ClassPin is a entirely portable tool, as it works by just pre-loading a binary, and works with minimal impact on its execution, as it is not based on binary analysis neither affects default allocator's policies and strategies. Last but not least, ClassPin defends such attacks not just by complicate their construction, but by eliminate completely the possibility for attackers to hijack class pointers.

Contributions.

1. We designed, implemented and evaluated ClassPin, a protecting mechanism against exploitations through use-after-free vulnerabilities for Objective-C, based on VTPin [37], a similar tool works on C++ binaries. Although that we used the same idea and the equivalent logic for defending the problem, we faced numerous difficulties during the implementation, due to major differences between the two languages and the ways they manage the memory and implement the object-oriented programming environment.

2. ClassPin is fully portable as it works directly on Objective-C binaries, does not based on binary analysis and does not interfere with the allocator's policies and strategies.

3. ClassPin is evaluated using Safari, the default web browser in MacOS platforms which is written in Objective-C. We used popular web browser benchmarks that heavily stress the capabilities of web browsers. As a result, the ClassPin experiences 0.38% -1.93% average memory overhead and 21.12% - 28.43% runtime overhead.

Chapter 2

Background

Contents

2.1 Software Security

Software security [31] is a relatively new field of Computer Science. Software exploitation problem is a serious and vital aspect of Computer Security, as intruders manage to hack into systems by exploiting software vulnerabilities. Especially, with the internet-enabled application growth and the constant addition of complexity and extensibility in software. The Software Security field's main idea is developing secure software that continues to retain the confidentiality, integrity and availability under malicious attacks. More specific, the developers must to be aware and understand the common threads and use right engineering techniques from the early stages of software developing to design and implements software free of bugs and vulnerabilities.

2.2 Application Security

According to Institute of Electrical and Electronics Engineers (IEEE) [7], Application Security [31] deals with the securing of software after their completion in contrast with Software Security (Section 2.1) which integrates the concept of security during the software's development. Although that implementing a secure software is much more easier than protecting one after its already completed, in many occasions revisioning and rewriting a program may be infeasible. So, there is a need for securing applications after their built with protections like Sandbox [27, 35], Control Flow Integrity (CFI) [20], Address Space Layout Randomization (ASLR) [34], Data Execute Prevention (DEP) [21] etc., or with the use of other applications specialized in a specific type of attack like the one we implement in this thesis dissertation.

A simple example shows the difference between Software and Applications Security is:

- *Application Security*: preventing buffer overflow by detecting malicious HTTP traffic.
- *Software Security*: Patch the source code to be completely impossible to create a buffer overflow.

2.3 Objective-C

Objective-C is an extension of C programming language with object-oriented constructions. Dynamic dispatch of function call is implemented for all function calls, in comparison with C++ which only enable this mechanism in virtual object occasions. In C++, VTable pointers are used at runtime to invoke a function implementation where the proper function is not known at compile time. However, every time a function is called in Objective-C a message is sent by calling a predefined function (*msgSend*), which locates and executes the right function implementation. These messages contain a receiver object, a selector and arguments, if there are any. The receiver object is a pointer to the object which calls a function and the selector is the name of the function to be called. In that point, the appropriate implementation might be found in object's class method list, in object's parents' classes method list or in a cache which saves previous calls for the same

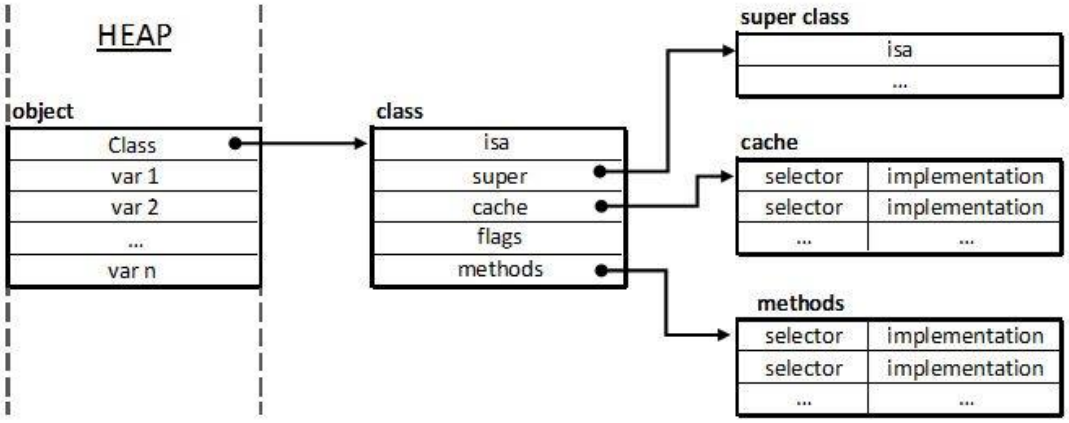


Figure 2.1: The memory layout of objects, classes, caches, and methods implementation.

object. Also, the class might provide the ability to forward the messages to other objects or dynamically respond to new messages in the runtime. However, in the compile time the object's class, and accordingly the all the other fields illustrated above, may not be yet known. Therefore, the compiler attaches a class pointer to each object's instance as it shown in Figure 2.1, so as to resolve the message dispatch's target in the runtime. As shown in Figure 2.1, super pointer contains the address of class' parent class or null if it is in the top of hierarchy, while the cache and methods pointers point to lists of selectors and the respectively function pointer for each implementation.

2.4 Class Pointer Hijacking

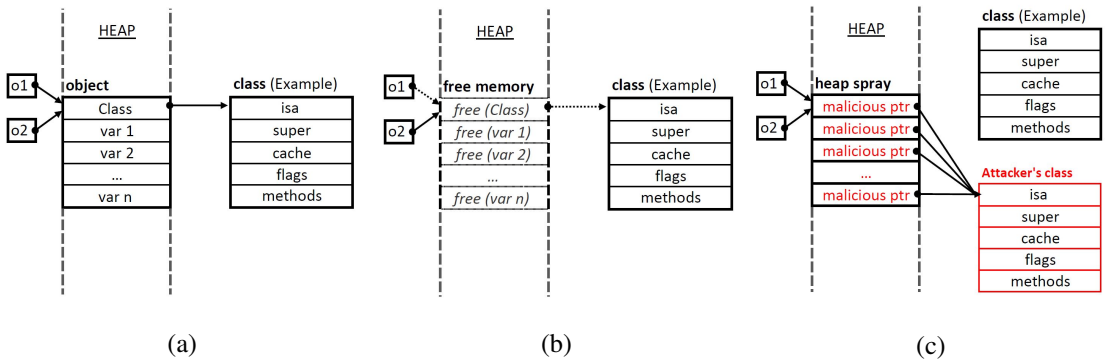
In modern days, software exploitations usually contain a step where use-after-free vulnerabilities are abused, because defenses in place fail to protect the process against these types of attacks. Exploiting use-after-free vulnerabilities is a process of actions which difficultly consider as fraudulent, as it is not involve memory corruption, such as writing over memory bounds or overwriting process' data structures (*e.g.,* override return address in the process' stack). The main idea of use-after-free exploitations is to utilize pointers that point to freed memory still accessible from a process. These pointers are created after the destruction of an object without modifying the value of the pointer which pointed to it, in that way so it still points to deallocated memory. After that, the system's allocator may reallocate the same memory area and filled it with completely different data as needed.

```
1  @interface Example: NSObject
2      - (void)foo;
3  @end
4
5  @implementation Example
6      - (void)foo {
7        NSLog(@"I am an object");
8      }
9  @end
10
11 int main(int argc, const char * argv[]){
12     Example *o1, *o2;
13     /*...*/
14     o1= [Example alloc]; //allocation of an example object
15     /*...*/
16     [o1 foo];
17     o2= o1;
18     /*...*/
19     [o1 release];   //deallocation of the example object
20                    //o2 is now dangling pointer
21     /*...*/
22     [o2 foo]; //use-after-free trigger
23 }
```

Figure 2.2: Example program to illustrate how use-after-free vulnerabilities can be abused for class hijacking. In line 12 two `Example` class pointers are declared, namely `o1` and `o2`. In line 14, an instance of `Example` class is created and o1 points at it. . After that, in line 17 `o1` and `o2` point to the same location, and at line 19 `o1` is deleted. The allocator frees all space occupied by the `Examples`instance, but `o2` still points to the location that `o1` was pointing at. If `o2` is accessed (line 22) the program behavior is undefined.

Now, if the program try to dereference this pointer, which is called dangling pointer, the behavior of the program will be unpredictable. Therefore, if an attacker manage to insert his data in the freed memory and then cause dereference of a dangling pointer which points to this space, he can carefully adapt the dereferenced data and control the data flow of the running process. While this attack can be achieved through any dangling pointer, in Objective-C usually class pointers are chosen because they are used to trigger indirect calls, as explained in Section 2.3, and consequently they are tremendously useful for changing the control flow of a program.

Figures 2.2 and 2.3 demonstrate the mechanics of a *class hijacking attack*. As it can be clearly seen in Figure 2.2, a class definition, `Example` class, is declared and implemented in lines 1 to 9. In line 12, the program declares two pointers of type `Example` namely `o1` and `o2`. Later in line 14, a new `Example` object is allocated and `o1` redirected to point on



(a) (b) (c)

Figure 2.3: Class hijacking mechanics. For the example program of Figure 2.2, we present, in (a), the memory layout of the instance that `o1` points to. The first 8 bytes are containing a class pointer, which points to the `Example`'s class, followed by the respective object's data. The `Example` class is stored in read-only memory and contains pointers to the parent's class, the cache and the methods of `Example`. In (b), we demonstrate what happens if `o1` is deleted, and in (c), how the attacker can abuse the dangling pointer (`o2`). An attacker can *spray* memory with the contents of a phony class pointer that points to data under the attacker's control. This data can be intentionally resemble a valid class with valid method implementations in such way, so that if dereferenced through a dangling pointer (Figure 2.2, line 22), then a forged *method* will be executed (*e.g.,* `mprotect`), instead of the one the original program called.

it. So, when the function `foo` is called in line 16 the *msgSent* function is invoked with `o1` as receiver argument, and the class pointer, which points to `Example` class, is used in order the right implementation to be found. Respectively in figure 2.3(a), the memory layout of the object that is pointed by `o1`, is illustrated. As shown, the first 8 bytes contain the address of the `Example` class, which is used as explained in Section 2.3, followed by object's data. Notice that super class, cache and methods pointers shown in Figures 2.1 and 2.3 are stored in read-only memory so they cannot be hijacked. Unfortunately, the class pointers are located in writable region as all objects instances are allocated in heap, stack or global data sections, so it may be overwritten.

Going back to Figure 2.2, `o1`'s value is copied to the `o2` in line 17 and subsequently in line 19 `o1` is deallocated. During the deallocation of `o1`, the memory that `Example`'s instance occupied marked as free and it can be reused from the allocator for future needs. Also, depending on the system's allocator, the contents of the free area and `o1` may be zeroed or left as are. However, `o2` still points to where the `o1` was pointing and where the class pointer was located, so `o2` is now a dangling pointer. Essentially, at that moment the

memory layout is illustrated in Figure 2.3(b). At this point, if `o2` is accessed, like in our example in Figure 2.2 in line 22 where the foo function is called, the program's behavior is undefined. Technically, the execution will be continued in three possible ways:

a. if the freed memory is zeroed or the class pointer points to a not valid address the program will crash.

b. if the data in the freed memory has not been changed, the implementation of `foo` function will be invoked.

c. if the freed memory has filled intentionally with data in that way that the class pointer points on another class, arbitrary code will be executed.

The scenario (c) is the one which attracts more interest for exploitation purposes. An attacker may *spray* [38] memory with malicious contents, specifically fill the memory with pointers to a class that is under his control, as demonstrated in Figure 2.3(c). Therefore, in case of a function call with the receiver of the message be a dangling pointer, a function of the attacker's class will be invoked. Notice, that the attacker is not able to inject code, as we assuming that the injection is done in not-executable pages [21], but in this occasion he writes memory addresses in this area. For example, he can insert data in that way so when a function is called the `mprotect` function be invoked, which, once called, modifies the permissions to executable in memory areas where the attacker has write privileges. Also notice that, the attacker's class can be implemented in that way so it forward any messages with any selector to a specific function implementation. So, in case of vulnerable process, such as the one illustrated in Figure 2.2, the attacker may abuse the memory like is shown in Figure 2.3(c), and hijack the control flow.

Chapter 3

Methodology

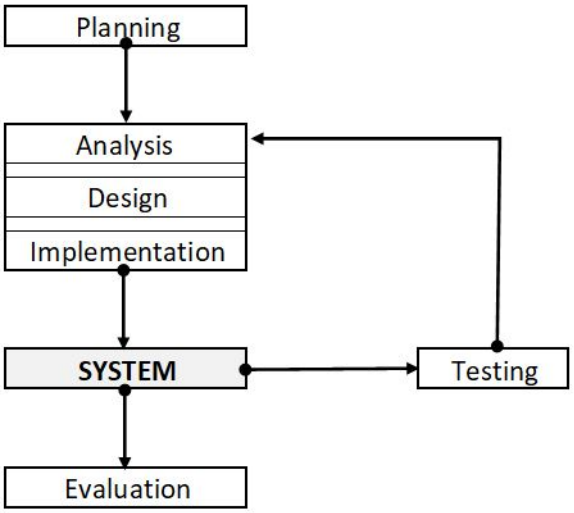


Figure 3.1: Extreme Programming Methodology model.

A software development methodology is a framework that is followed during the designing and construction of a new software application, and it is used for structuring, planning and controlling the developing process. There are several methodology models and each one has its own known advantages and disadvantages. For this thesis dissertation, we chose Agile Software Development and specifically Extreme Programming model as it seems to be the most suitable. Extreme Programming methodology refers to the approach that a system is created in iterations and each iteration includes implementation testing, feedback and planning. It is flexible, simple, helps the new application to improve through continuous testing, can be used for short period projects and in small teams which made it the appropriate methodology to use.

As shown in the Figure 3.1 the model includes two main phases: planning and iteration stage. In the planning phase, we research and understand the security issues and how they apply in Objective-C. Also, we investigated solutions of the respective problem in other programming languages or with different approaches. Finally, the system's development plan and its requirements were set.

The second phase is divided into three other phases: Analysis, Design and Implementation. In each iteration, a new prototype of the system was created and tested. Their defects were analyzed in a weekly meeting and a new system was designed and implemented. Finally, when the system was meet the specifications and requirements we proceed to evaluation and reporting phase.

Chapter 4

Architecture

Contents

ClassPin intent to protect Objective-C programs from being exploited through use-after-free bugs and class hijacking. The solution has to interfere with the running process with the least possible impact and has low time and memory overhead. Consequently, it is necessary to meet the following requirements:

- No need for source code, recompilation or debugging symbols
- No analysis, disassembling or patching of the binary
- No interfere with allocator's policies or strategies.

The main idea of our solution is simple, however, the implementation of a system which applies it, may be complicated enough (Section 5). The majority of use-after-free attacks in Objective-C programs are based on class pointer hijacking through intentionally and strategically (re)allocation of freed memory. So, instead of searching and protecting each dangling pointer [23], we ensure that all class pointers are always valid. Hence, they cannot be hijacked by future malicious reallocations. Based on this, we were able to promote a fully portable and generic solution for defending software exploitation through use-after-free vulnerabilities and class hijacking. ClassPin handles all deallocations of the program and ensures that all class pointers will be preserved. Specifically, ClassPin resolves the class pointer in the space that going to be freed, preserves it and overwrites it

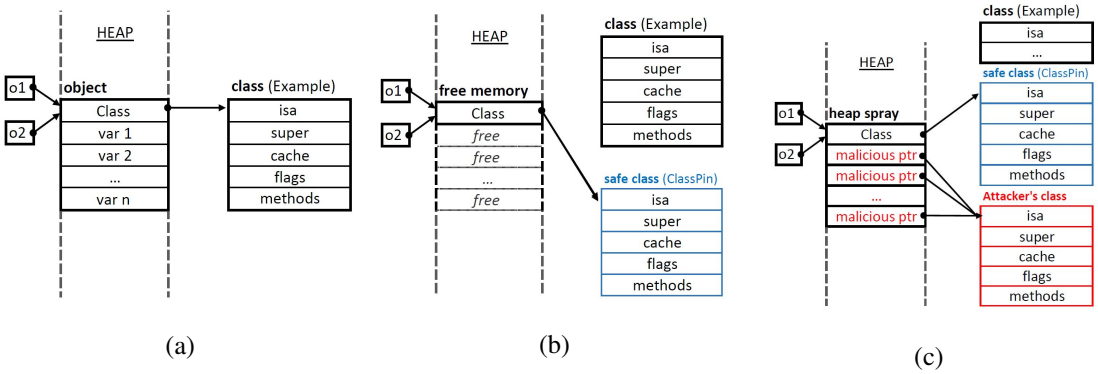


Figure 4.1: Class Pointer hijacking prevention when ClassPin is enabled, using as example the program illustrated in Figure 2.2. In the beginning, in (a), the two pointers, `o1` and `o2`, point to `Example`'s instance (Figure 2.2, line 17). Later, in (b), the `o1` is freed (Figure 2.2, line 19) and the ClassPin frees all the space occupied by the `Example` instance except of the class pointer. In addition, the ClassPin redirects the class pointer to point on a safe class controlled by itself. Finally, an attacker may spray the memory with malicious pointers, as in (c), however the class pointer is preserved and cannot be hijacked. Also, if a dangling pointer (`o2`) try to invoke a method, as shown in Figure 2.2at line 22, a safe method of the ClassPin's safe class will be invoked and the call will be tracked.

in that way it points to a safe class under its control, while it releases all the other memory occupied. Essentially, it prevents the class pointers' memory from being freed so their space is unavailable for future allocations. Although dangling pointers may continue to exist in the program, they are effectively neutralized and they are harmless if they are triggered. In addition, if ClassPin is in place and a dangling pointer is triggered, not only the program will not crash but also it can help to detect the dangling pointer, so it can be eventually patched.

Figure 4.1 shows how ClassPin handles the memory, using as example the vulnerable program demonstrated in Figure 2.2. In the beginning, the two pointers, `o1` and `o2`, point to the class pointer of the `Example`'s instance (Figure 2.2 , line 17), and respectively that points to the `Example`'s class. The Figure 4.1(a) shows the memory layout of the object and its class at this point. In line 19 (Figure 2.2), the `o1` is deleted, and the ClassPin takes control which frees all the memory occupied by the `Example`'s object except of the class pointer, as demonstrated in Figure 4.1(b). Following, it overwrites the class pointer with the address of a safe class controlled by itself. As illustrated in Figure 4.1(c), an attacker may spray the heap with phony pointers; however, the space occupied by the class pointer is never freed, so it cannot be hijacked. Finally, when the dangling pointer (`o2`) tries to

call a method (Figure 2.2, line 22), the message will be forwarded to a safe method of the ClassPin's class, resulting the invoke of a safe method which can help to log the call.

The security provided by ClassPin unfortunately has a cost, as memory and time overhead are introduced. Extra memory is needed because we preserve all class pointers, and extra time is consumed because we need to resolve which pointers have to be preserved. We managed to minimize the time overhead with a low complexity algorithm which rapidly distinguishes object's destructions from other memory deallocations. This algorithm is explained later in Section 4.1.

Regarding the memory overhead, for each object destruction, only the class pointer (8 bytes) survives. Unfortunately, class pointers need to be preserved in all object destructions, in comparison with the respective solution provided by VTPin, which preserves the VTable pointers in virtual objects only. Moreover, several applications, like web browsers, are based on short-lived process models. In other words, those programs fork new processes, to execute some procedures, which are terminated after a while. In such cases, there is no practically memory overhead as the preserved pointers will be freed at the process termination. However, the preserved pointers could occupy a large amount of memory in long-lived processes. Consequently, we propose a garbage collector (GC) in Section 7.1 as a solution. The garbage collector can periodically scan the memory for potential dangling pointers and free the preserved class pointers that are not referenced by any of them. Thus, we still retain all class pointers that are possibly referenced from other memory locations in order to keep the process secure from use-after-free attacks. Notice that, garbage collector could be a costly operation, although, it could free a significant amount of memory as the majority of the preserved class pointers will not be referenced (*i.e.,* we assume that the majority of programs does not contain a vast amount of dangling pointers). Furthermore, the GC, unlike the traditional garbage collectors [22], is used just to free class pointers that there is no need to be preserved anymore, so it can be invoked rarely and with low performance overhead on the running process. As it can be clearly be seen in Section 6, we estimate the memory overhead of ClassPin for small example programs and even with the use of complicated benchmark suites that heavily stress the capabilities of web browsers.

4.1 Object Resolving

ClassPin secures programs from use-after-free attacks without needing access to the source code or the binary and without modifying the system's allocator. As a matter of fact, it intervenes in all `free` calls of the process, and handles the deallocation only when it is relative to an object. Therefore, it has to resolve if the chunk of memory, that going to be freed, compose an object or not. In the following, we simply explain the main algorithm of this procedure. The technical details for implementing the algorithm and the ClassPin prototype are provided in Section 5.

Free is called with a pointer (`ptr`), to the first byte of the memory to be freed, as a parameter. ClassPin passes the pointer through some stages until it concludes to proceed to the program allocator's `free` function or to handle the memory as an object. Firstly, the `ptr` is checked if is valid (*i.e.,* not null, Surprisingly a lot of `free` calls are called on null pointers), and if it is not, does nothing and return to the normal program's execution. Later, the `ptr` is passed through two stages until is surely resolved to its category. Each stage corresponds to a heuristic which distinguishes pointers that point to object's relative memory from them that are not. We use two main heuristics:

1. *Memory Page's Permissions*: Check if the class pointer belongs to a writable or executable memory page by examining the permissions of all memory regions of the running process. All classes are stored in same type region , so if the class pointer does not point to one of them then the `ptr` is not associated with an object and it can be simply freed using the program's default allocator. Finding the permissions of all memory regions is an easy and fast procedure, as all shared libraries are mapped linearly and ASLR or fine-grained randomization [26, 28, 33, 39] has no impact.

2. *RTTI*: We extract the first 8 bytes of the memory block that is pointed by the `ptr`, and we treat it like is a class pointer. Specifically, we use run-time type information functions provided by Objective-C [12] to determine the class type of the assumed class pointer. If the class type is discovered successfully it means that the `ptr` is *definitely* related to an object. On the other hand, in case of memory block that is not associated with an object, there is high possibility for a fault, due to reference to unmapped memory. Such faults must be handled from the ClassPin with no impact to the running process, as we further discuss in Section 5.

Due to the fact that we implement the ClassPin for two different operating systems

(*i.e.,* Linux, MacOs) with great differences between them, the algorithm of each prototype slightly differ. However, the main idea behind the algorithm of both prototypes is the one we discuss above. We illustrates the differences of the two prototypes and the way each prototype adapts the algorithm in Section 5.3

Chapter 5

Implementation

Contents

We implemented two prototypes of ClassPin one for each operating system, Linux and MacOS. The two prototypes are both written in C/Objective-C and target Objective-C binaries on 64bit systems. They are based on the same main idea, however, there are some variations between the two prototypes due to the differences in the way the compilers handle the Objective-C programs in each operating system. In this chapter, we further explain the implementation details for each prototype, and also, we provide and discuss some technical aspects that need to adjust in order to use ClassPin on each platform.

5.1 Portability Requirements

ClassPin is fully portable as long as the following requirements are met.

1. *Hooking free:* ClassPin works by instrumenting each `free` call on a running process. Therefore, ClassPin have to be able to hook free either if the system's allocator is used, or if a custom allocator is in place [5].

2. *Realloc in the same address:* ClassPin works by preserving the class pointer of each object, that going to be freed, and deallocating the rest memory, that is occupied by it. Also, the class pointer has to be preserved in the exactly memory address is found in order to neutralize possible dangling pointers that point to this address. The ClassPin's porotypes are based on `realoc` function of the standard `glibc` allocator. In cases when the `realloc` is called with a new size, (*i.e.,* `n` bytes), smaller or equal to the space that is allocated, the `realloc` simply keeps the first `n` bytes and discards the rest [16] . If an allocator [25, 29] does not implement `realloc` in the same way and handles the reallocation by moving the `n bytes` memory block to a different address than the one that initially was allocated at, the ClassPin will not work. This happens because a probable dangling pointer still points to the original address of the object, and so an attacker is still able to fill it with malicious data and abuse the control flow of the program. However, this problem can be solved with a simply modification of the ClassPin, in that way that for each `free` associated with an object do nothing (*i.e.,* no realloc or free call). Also notice that in case of such modification the memory overhead will sharply increase, as for each object deallocation all the object's space will be preserved instead of just one pointer.

3. *Handling invalid memory accesses:* Objective-C offers some runtime functions [12] that gives runtime type information for pointers, objects and classes. ClassPin uses those functions to resolve if a pointer is associated with an object (Section 4.1). Hence, when the ClassPin tries to invoke a runtime function with a non-object argument, there is a high probability of unmapped memory to be touched. In that case a segmentation fault is caused. Therefore, ClassPin has to be able to recover from a `SIGSEGV` signal or touch unmapped memory without causing a `segfault`. Most platforms support handling `SIGSEGV` in user space, so ClassPin uses this technique.

5.2 Basic Components

ClassPin consists of two basic components: (a) memory map that contains all the memory regions of the process and their permissions and (b) a safe class where the preserved pointers are pinned when an object is deallocated.

5.2.1 Memory Map

One way to separate memory blocks associate with an object from those who do not, is to check the permissions of the memory pages where the object's components are allocated to. To achieve that, ClassPin keeps a memory map with the process's memory regions and their permissions. Specifically, the ClassPin maintains a table with the first and last address of each memory region and their page's permissions. Also, due to the fact that there are not overlapping regions, the search in the memory map can be done efficiently. So, for each pointer can be quickly resolved in what type of memory page is pointed at (*i.e.,* execute or writable). ClassPin reads the memory region's information at the initialization of it. In addition, we hook `dlopen` in order to check when the memory regions of the process are changed and the memory map needs to be updated. In Linux, ClassPin collects these pages by reading the `proc/self/maps` file of the running process. In MacOs this file does not exist, so we create it using a system call with `vmmap` command.

5.2.2 Safe Class

The first time an object is deallocated, the ClassPin allocates a special safe object. The class of this special object is the safe class in which all preserved class pointers will be point at in all the next objects deallocations (Figure 4.1). Furthermore, the safe class implements a forwarder which forwards all `msgSend`, regardless to the selector, to a specific function in the safe class. Therefore, any dangling pointer, if triggered, invokes that function and the attacker no more has the ability to abuse the control flow of the program. Also, this special function prints information for the dangling pointer in order to help the administrator to find it and correct the bug. Finally, the execution of the program does not terminate, since the dangling pointer is no dangerous any more, because the termination of the program in such cases would provide an advantage for attackers to use the ClassPin's functionality against it and produce denial of service attacks.

5.3 Differences between Linux and MacOS Prototypes

The compiler in each operating system, Linux and MacOS, handles differently the Objective-C programming language .In this Section we discuss the differences between the two operating systems and the way ClassPin handles them. The compiler in each operating system handles differently the Objective-C programming language.

1. *Extra bytes:* Although that the memory layout of an object is the same in both operating systems, in Linux the `free` function is called with a pointer that points 16 bytes above the object as argument. ClassPin's Linux prototype makes the all the calculations needed in order to handle the class pointer, which is on the first 8 bytes of the object. However, due to the way the `realloc` works 24 bytes are preserved instead of 8 (pointers size).

2. *Memory Region:* In MacOs the classes are allocated in read-only memory pages (*i.e.,* executable and not writable), while in Linux classes belong to writable memory pages (*i.e.,* not executable). Therefore the two prototypes check for the appropriate memory pages' permissions respectively.

3. *Class Pointer Value:* In Linux, the class pointer value is changed during the destruction of the object before the function `free` is called. In order to be able to use the heuristics we illustrate in Section 4.1(RTTI and Memory Page's Permissions), the ClassPin needs to know the original address the class pointer was pointed to. We analysed the deallocation procedure and we found out that the class pointer value is changed to `0xdeadface` using the `object_setClass` function. Therefore, ClassPin prototype in Linux, places a hook to the `object_setClass` function and preserves the original function to the class pointer if the new value that going to be set is equal to `0xdeadface`.

4. *Loading Memory Map:* In MacOS, there is not a file which contains the current mapped memory regions like Linux, so ClassPin has to create it with a call to `vmmap`. This system call cannot be executed in the initialization phase of ClassPin because this procedure contains `free` calls, and subsequently the system could crash or an endless loop would be created. Consequently, ClassPin prototype on MacOS, places a hook to the `release` function and loads the memory maps in the first call of it. Also, the ClassPin's special `free` function invokes the allocator's default `free` function until the memory maps are loaded. So, the memory maps are loaded exactly before the first probable object's destruction in order to set in place the ClassPin's `free` function and consequently the protection.

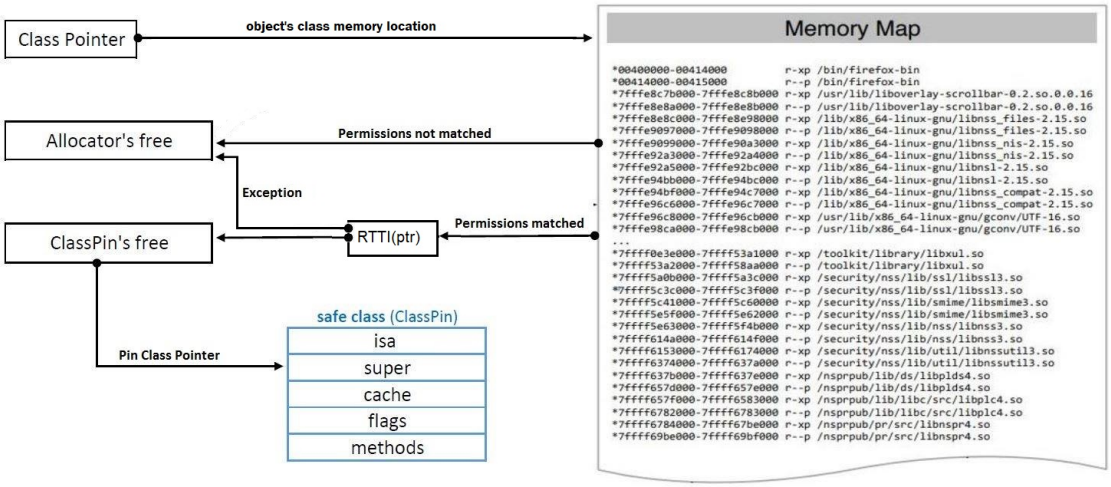


Figure 5.1: ClassPin prototype overview. For every pointer that going to be freed, ClassPin resolves if the pointer is relative to an object or not (Section 4.1). Firstly, the memory map is used to check the permissions of the object's class, assuming that the first pointer is pointing to one. Later, the pointer passed through the phase, where ClassPin supposes that the pointer is associated with an object and treats its like one by using runtime functions. If the pointer pass the two stages then ClassPin's free invoked otherwise the default free function is called. ClassPin's free preserves and pins the class pointer to a safe class and deallocates the rest memory occupied by the object.

5.4 Object Resolution

Figure 5.1 shows a high-level overview of the ClassPin's algorithm for object's resolution which is described in Section 4.1. Given a pointer to be freed, ClassPin assumes that the pointer is associated with an object and extracts the supposed class pointer. Then the permissions of the memory region where the class pointer point to are rapidly checked. To manage that ClassPin maintains a memory map as described in Section 5.2.1. Also, each prototype's check is different as illustrated in Section 5.3. So, the ClassPin very quickly resolve pointers that point to memory pages with different permissions and does not proceed to the next algorithm stage which is little more costly.

To the next stage of the algorithm, ClassPin uses a RTTI function, `object_getClass` which returns the class of the given instance. In that point, ClassPin can mark a pointer as not relative to an object in two occasions. First, if an exception was created during

the RTTI function's call, and second, if the returned value from function is not valid. However, in case of first, the ClassPin has to handle the exception. So, before calling any RTTI, ClassPin saves the current state of the calling environment [18] with the use of `setjmp()` function. Also, a custom exception handler is called when a `SIGSEGV` signal occurs, which is calling `longjmp()` and returns the control to the same point the `setjmp` saved. Notice that, during the RTTI function's calls more `free` calls occur which must be handled carefully without creating an infinitive loop.

5.5 Class Pointer Pinning

Once ClassPin finds pointers that are associated with an object a special deallocation procedure is invoked, otherwise the default allocator's `free` function is executed normally. In the case of the first, the class pointer should be preserved and pinned to a safe class (Figure 5.1, while the rest of the memory should be normally freed. ClassPin shrinks the object to 8 bytes by using the `realloc` function. By this, the class pointer, which is at the first 8 bytes of the object, is preserved at the original memory address while the rest of the object is deallocated. Finally, the value of the class pointer is changed to point to the safe class provided by ClassPin. Class pointer hijacking is not possible anymore, as any dangling pointer trig, will invoke a safe method under the ClassPin's control. Also, the value of the class pointer cannot be overwritten by future allocations as this space is never freed.

5.6 Compiling and Running

ClassPin tool is a software compiled into a shared library and loaded along with the binary that going to be protected.In this section we provide extra details for compiling and running the two ClassPin's prototypes.

5.6.1 Linux

In Linux operating system, the compilation can be done either using GCC or LLVM Clang compiler. To create a shared library (`.so` extension) use `-fPIC -shared` flags during the

compilation. In order to run ClassPin along with a binary the shared library, which was created, should be loaded before other libraries, so the ClassPin's functions overwrite the default ones. To preload a shared library in linux, the `LD_PRELOAD` can be used.

5.6.2 MacOS

In MacOS operating system the compilation can be done using the system default LLVM CLANG compiler. To create a dynamic library (`.dylib` extension) use `-dynamiclib` flag during the compilation. Also, the flag `-framework Foundation` has to be added as the ClassPin use it. The dynamic library should be loaded first similarly to Linux. However, in order to preload a dynamic library in MacOS is slightly more complex than in Linux. Firstly, the respective command `DYLD_INSERT_LIBRARIES` is used instead of `LD_PRELOAD`. Secondly, the environment variable `DYLD_FORCE_FLAT_NAMESPACE` has to be set to 1 in order to ignore any two-level namespace bindings [9]. Moreover, the System Integrity Protection (SIP) should be disabled as it prevents the dynamic library from preloading. The Systems Integrity protection on a Mac can be disabled from a terminal in a recovery mode using the command `csrutil disable` [6] . Finally, MacOs prevents the preloading when the binary and the dynamic library images have different signatures. We can bypass this protection by signing the binary with the same signature as the ClassPin.

Chapter 6

Evaluation

Contents

In this section, we illustrate how ClassPin success to secure Objective-C binaries and present the ClassPin's overhead in terms of performance and memory. Both Linux and MacOS ClassPin's prototypes were carried out using virtual machines, Oracle Virtual Box v 5.2.8, on Windows 10. The Linux virtual machine was set up with a 2.5GHz quad-core Intel Core i5-7500U CPU, 4GB RAM and the operating system was 64bit Ubuntu Linux v 16.04 LTS. The MacOS prototype was tested on 64bit macOS High Sierra v 10.13 armed with 2.7GHz CPU processor and 5GB RAM.

Because there are no software written in Objective-C available for running on Linux based operating systems, the Linux ClassPin prototype was tested for the effectiveness and for the performance overhead with small programs we implemented. On the other hand, the MacOS ClassPin prototype was evaluated by running the Safari web browser, which is the MacOS default browser and is written with Objective-C, and by using popular browser benchmarks.

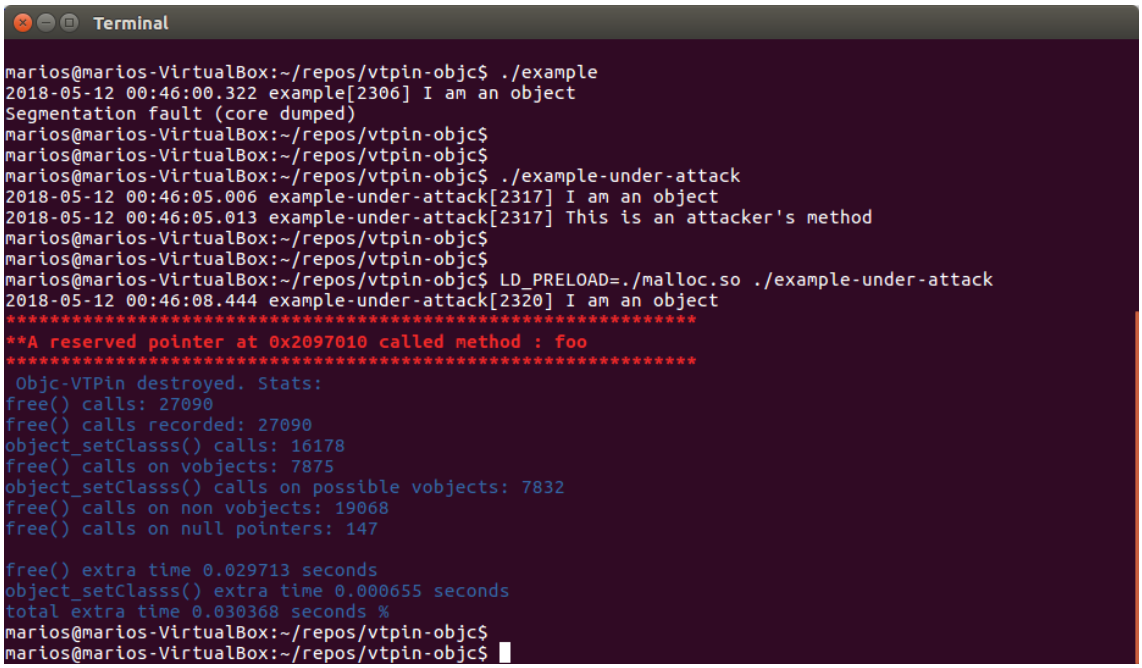


Figure 6.1: Execution of the example's program of Figure 2.2 in Linux environment. Three different executions are shown: a) `./example` **execution of the example program without protection, b)** `./example-under-attack` **execution of the example program slightly modified to simulate use-after-free attack by abusing** (`o2`) **dangling pointer, c)** `LD_PRELOAD=./malloc.so ./example-under-attack` **execution of the same binary like in (b) but with ClassPin enabled.**

6.1 Effectiveness

We assessed the security effectiveness of the ClassPin's prototypes by using the sample program we illustrated in Figure 2.2. Figure 6.1 presents what happens in the execution of the example program in Linux platform; (a) without protection, (b) without protection under and (c) with the use of Linux ClassPin tool. Respectively, Figure 6.2 shows the execution in MacOS platform for the same scenarios. Notice that we did not actually attack the program, but we simulated the attack.

In scenario (a) we run the example program presented in Figure 2.2 with no protection enabled (`./example`). In line 16, the `foo` function is called which prints a message to the console. Then in line 22, the `foo` function is called again but at that point from a dangling pointer (`o2`) as the `Example` object had freed (line 19). In MacOS (Figure 6.2) the same message as before is printed, while in Linux (Figure 6.1) the program crashes. This is happening because during the deallocation of the object in MacOS the values of the freed memory were not changed, so the `foo` function is invoked. However, in Linux the

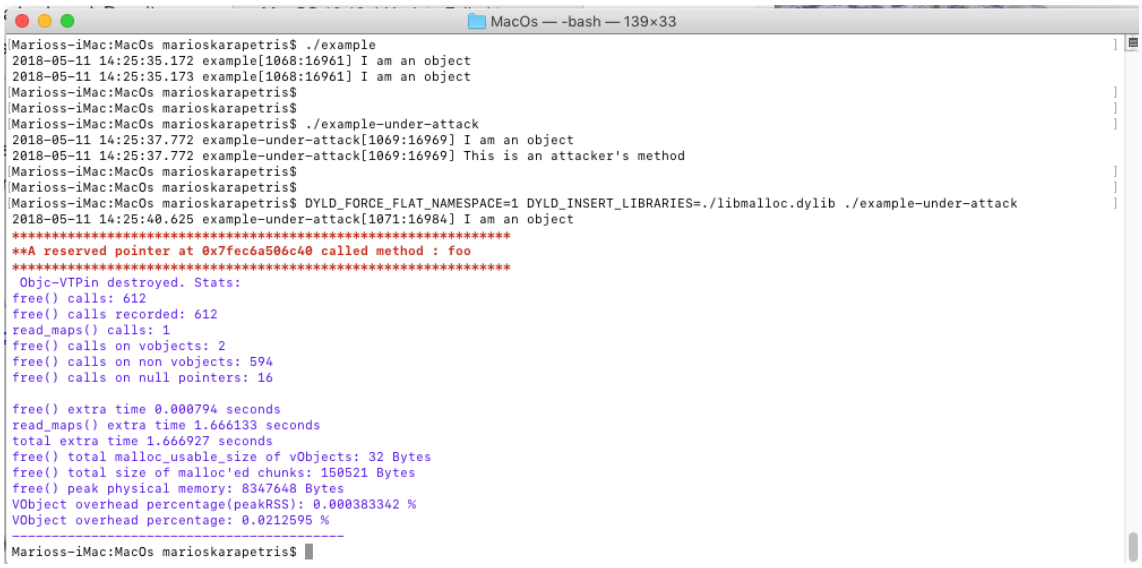


Figure 6.2: Execution of the example's program of Figure 2.2 in MacOS operating system. Three different executions are shown: a) `./example` execution of the example program without protection, b) `./example-under-attack` execution of the example program slightly modified to simulate use-after-free attack by abusing (`o2` dangling pointer, c) `DYLD_FORCE_FLAT_NAMESPACE=1 DYLD_INSERT_LIBRARIES=./libmalloc.dylib ./example-under-attack` execution of the same binary like in (b) but with ClassPin enabled.

compiler changed the value of the class pointer to `0xdeadface`, therefore the program crashed because invalid memory is touched when the `msgSend()` is called.

In scenario (b) we simulated an attack to the example program of the Figure 2.2 (`./example-under-attack`. We managed to simulate a use-after-free attack by allocating an object of type `Attack` just after the deallocation of the `Example`'s object. Due to the fact that the compilers place the new object exactly to the same location the freed one was if their sizes are exactly the same, the program behaves as it would do if it was under a class pointer hijacking attack. As it can be clearly seen, in both operating systems, the attacker's function is called which in our example just prints a message to the console.

Finally, in scenario (c) we present the effectiveness of the ClassPin tool by preloading ClassPin's shared library in the binary of scenario (b) execution. Similarly to scenario (b), the two operating systems have the same results. However, now the class pointer was preserved and was pinned to the safe class. So, in line 22 when the dangling pointer calls the `foo` function, the safe method of ClassPin's class is invoked and prints (red color) the function name and the memory location of the dangling pointer in order to help the developer find it and fix the bug.

Statistic	Value	Percentage
Total `free` calls	20,008,555	100%
Unhandled `free` calls	0	0%
`free` calls on `null`	99	0.0005%
`free` calls on non-objects	10,008,412	50.02%
`free` calls on objects	10,000,044	49.98%
`object_setClass` calls	20,000,108	100%
`object_setClass` calls on possible objects going to be freed	10,000,007	49.99% (99.99% of objects)
	Time	**Overhead**
Total time (ClassPin disabled)	2.02 sec.	-
Total time (ClassPin enabled)	5.05 sec.	60%
`free` extra time	0.99 sec.	19.6%
`object_setClass`	0.76 sec.	15.05%
Total extra time	1.75 sec.	34.65 %

Table 6.1: Statistics for evaluation of ClassPin Linux prototype. The statistics were calculated by running an example program which in a loop of 20 millions iterations were allocated and freed 10 millions objects and 10 millions string buffers.

6.2 Linux Prototype

The security that ClassPin provide comes with a cost. In this section, we present and discuss the cost of ClassPin as it was measured using the Linux prototype. The evaluation of ClassPin on Linux was estimated using an example program we implemented because there is no known software written on Objective-C language that running on Linux machines. Specifically, we implement a program in Objective-C that uses a loop of 20 million iterations to allocate and then release 10 million objects and 10 million string buffers. The example program is run firstly `normally` and then by preloading the ClassPin shared library to enable the ClassPin's protection.

Table 6.1 presents the statistics obtained during the executions. Firstly, we illustrate the `free` and `object_setClass` function calls divided into seven main categories, as they

were counted by the ClassPin. The example program invoked 10 million times the `free` function with a pointer to a string buffer as an argument and released an object, which is also calling the `free` function, another 10 million times. As it can be clearly seen, the ClassPin recognize the if the free is associated with an object and separated the calls to calls on object and calls on non-objects. The free calls are just above 20 million because the `free` function is also used during the initialization and termination of the program. The calls of `object_setClass` are shown too. The `object_setClass` is used to set the value of the class pointer on an object, and as we mentioned in Section 5.3 is changing the class pointer of an object going to be freed to `0xdeadface`. The `object_setClass` function was called just around 20 million times, as it is called once at the allocation phase and once during the deallocation. Moreover, the ClassPin counts the calls where the class pointer is set to `0xdeadface` as possible objects going to be freed.

The table also shows the performance overhead of the ClassPin. The runtime overhead is calculated in two different ways. Once by using the `time` command provided by Linux systems, and the other by using times calculated by the ClassPin during the execution. The first method shows 34.65% and the second around 60% overhead. The big difference in the results is due to that the `time` command is also counting the time needed to preload the shared library and the time ClassPin takes to retain and print the statistics. Finally, is shown that the `free` causes the 19.6% of the overhead and `object_setClass` 15.05%. In a real scenario of implementing this solution to a system, the compiler could be modified so it does not call the `object_setClass` function during the deallocation, as the change in class pointer value is no longer needed. Additionally, there will be no need for statistics keeping that consuming extra time. As a result, the runtime overhead would be lower than 20%.

6.3 MacOS Prototype

In this section we measure the performance and memory overheads of the ClassPin's prototype on MacOS. The calculations are made by using Safari web, the MacOS platform default web browser. Moreover, we push safari to the limits by using widespread web browser benchmark suites. Namely, we use `ARES-6` [3], Octane [13], Spedometer [19], MotionMark [11] and JetStream [8].

Benchmark	Calls	Unhandled	null	Non-object (freed)	Object (preserved & pinned)
ARES-6	5,949,071	84 (0.001%)	593,302 (9.97%)	5,209,164 (87.56%)	146,521 (2.46%)
JetStream	16,198,052	89 (0.0005%)	1,598,916 (9.87%)	14,247,810 (87.96%)	351,210 (2.16%)
MotionMark	9,746,480	70 (0.0007%)	970,986 (9.7%)	8,646,199 (88.71%)	129,225 (1.32%)
Octane	3,626,358	28 (0.0008%)	369,143 (10.17%)	3,190,227 (87.97%)	66,960 (1.84%)
Speedometer	9,892,087	92 (0.0009%)	978,284 (9.88%)	8,693,270 (87.88%)	220,441 (2.22%)

Table 6.2: Distribution of `free` calls for popular benchmarks run on Safari under ClassPin's MacOS prototype protection.

6.3.1 Deallocation Calls

Table 6.2 illustrate the calls' distribution of the `free` function, as they were counted by the ClassPin for the web browser benchmarks. Firstly, the calls are presented in two categories, handled and unhandled. The unhandled calls are the ones were made before the ClassPin's initialization. For example, during the creation of the memory map (before that ClassPin cannot check the pointer going to be freed for association with an object. Then `free` calls are distributed to three categories:

a. *Calls on* `null` : the ClassPin returns to the execution without further actions.

b. *Calls on non objects* : the ClassPin invokes the allocator's `free` function to handle the deallocation.

c. *Calls on objects* : the ClassPin preserves and pins the class pointer to a special class, under its control, and then invokes the allocator's `realloc` function to shrink the object to the size of a pointer (*i.e.,* 8 bytes).

As it can be observed from the Table 6.2, almost 10% of `free` calls are called with `null` pointer as argument, which is too easy and fast for ClassPin to handle them. Moreover, the majority of calls, around 88%, in common programs is not associated with an object deallocation, therefore there is no memory overhead for them as they normally freed. Finally, the `free` calls which are referred to an object are 66,960 out of 3,626358 (1.84 %) for Octane, which is the best-case scenario, and 146,521 out of 5,949,071 (2.46%) for ARES-6, the worst-case scenario. These calls must be specially handled, so the less they are the less memory overhead occurs.

6.3.2 Memory Overhead

Binaries are protected from class pointer hijacking through use-after-free vulnerabilities by preserving the class pointer in each object deallocation. Therefore, is very important to measure the memory volume that remains occupied by the preserved pointers instead of released rather than being released as will be done without using the ClassPin.

To estimate the memory overhead, in each `malloc` call (`hook`) we record the size of every allocation to calculate the total malloced chunks as shown in the first column of the Table 6.3. Also we get the resident set size (RSS) [17], second column of the table, by using the `getrusage` function [4]. The RSS is always smaller than total malloc size because some parts of the memory are not used and are not located in the main memory (RAM). Although the actual memory overhead is the one calculated by using the total size of malloced memory, the memory overhead calculated by using the RSS is more objective because only the extra occupied memory in RAM is considered important. The freed objects are shrinked to the size of a pointer (8 bytes) during the destruction. However, the actual size they occupy may be larger due to memory fragmentation. We compute the actual memory size occupied by preserved pointers by using the `malloc_size` function [10], which returns the size of the memory block pointed by a pointer, and we present it in the third column of the table.

In fourth and fifth columns of the Table 6.3 we present the memory overheads, calculated as explained above, for the web browser's benchmarks. The `actual` average overhead computed using total malloced memory size is 0.38% while the `real` average overhead, computed using RSS, is 1.93%. Regardless that ARES-6 has higher percentage of `frees` on objects (2.46%) than JetStream (2.16%) as shown in Table 6.2, the memory overhead of the second is greater (ARES-6 0.33% - 2.05% , JetStream 0.67% - 3.58%). Octane has the least `free` calls on objects (1.84%) and the least memory overhead (0.16% - 1.04%).

6.3.3 Performance Overhead

ClassPin instruments each free call and resolves if the pointer, that going to be freed, is relative to an object or not. Therefore, this procedure comes with a cost, runtime overhead. In this section, we estimate the extra time consumed by ClassPin MacOS's prototype in order to provide protection to Safari running the web browser benchmarks

Benchmark	Total malloc'ed Memory	Peak Memory Usage	Usable Size of Preserved Objects (maloc_size())	ClassPin Overhead (peak RSS)	ClassPin Overhead (Total malloc)
ARES-6	681,520 KB	111,412 KB	2,289 KB	2.05%	0.33%
JETSTREAM	815,044 KB	153,376 KB	5,487 KB	3.58%	0.67%
MotionMark	719,755 KB	157,272 KB	2,019 KB	1.28%	0.28%
Octane	645,095 KB	100,288 KB	1,046 KB	1.04%	0.16%
Speedometer	711,967 KB	205,672 KB	3,444 KB	1.67%	0.48%

Table 6.3: Memory Overhead for popular web browser's benchmarks run on Safari, under ClassPin's MacOS prototype protection.

Benchmark	Total time with ClassPin	Total time no ClassPin	Extra free time	Extra time for memory map	ClassPin Total extra time	ClassPin Overhead (extra times)	ClassPin Overhead (compare executions)
ARES-6	134 sec.	114.69 sec.	33.85 sec.	5.9 sec.	39.76 sec.	29.67%	14.92%
JETSTREAM	312.44 sec.	232.79 sec.	83.55 sec.	7.19 sec.	90.75 sec.	29.04%	25.49%
MotionMark	402.95 sec.	334.8 sec.	80.53 sec.	5.14 sec.	85.67 sec.	21.26%	16.91%
Octane	83.58 sec.	54.42 sec.	19.71 sec.	10.64 sec.	30.36 sec.	36.32%	34.88%
Speedometer	270.30 sec.	233.98 sec.	59.82 sec.	10.02 sec.	69.84 sec.	25.83%	13.43%

Table 6.4: Runtime Overhead for popular web browser's benchmarks run on Safari, under ClassPin's MacOS prototype protection. Two overheads are illustrated, the first is calculated from stats that ClassPin is measuring during the execution and the second is worked out from total execution times gotten using the `time` command provided from all linux based platforms.

suites mentioned above. The Table 6.4 illustrates the measurements taken.

Likewise in Linux evaluation 6.2, the runtime overhead is estimated in two ways, first by using the `time` command provided and second by using times calculated by the ClassPin. The difference in execution times shows an average runtime overhead of 21.12%, while the estimation of runtime average overhead with the ClassPin statistics is 28.43%. Notice that the ClassPin in MacOS has to execute a system call to create the file before reading it to retain memory maps, a procedure that causes an average overhead of 4.88%. Also, significant time is consumed due to statistics keeping from ClassPin.

Chapter 7

Future Work

Contents

7.1 Garbage Collector

In this section, we propose a garbage collector as an extension to the ClassPin. The idea is to search for preserved class pointers that they do not have reference on them, and release them. However, all the preserved pointers that possibly referenced from other memory location should be retained. As a result, the memory occupied from unnecessary preserved class pointers could be released. The garbage collector could work with low overhead by running it periodically as it does not need to be enabled in all time. Also, a lot of memory could be freed, as the majority of the preserved pointers are not expected to be referenced from other locations, as we assume that most programs does not contain large numbers of dangling pointers.

The main idea how to implement the ClassPin's garbage collector is discussed here. During the deallocation of an object, except from pinning the class pointer to a safe class, the address of the preserved class pointer should be saved in a data structure (*e.g.,* array, list, ...). When the garbage collector is invoked, it scans the heap, stack and global data sections of the process for addresses that are contained to the data structure mentioned above. If an address is matched, then possible dangling pointers exists, and should be marked. After the completion of the scan, the garbage collector could search the data

structure with the addresses of preserved pointers, and release all the unmarked pointers. Finally, this process can be aggressively parallelized by scanning with different CPU cores each memory region.

Chapter 8

Related Work

Contents

8.1 VTPin

VTPin [37] is a system that protects C++ binaries against VTable hijacking. VTable hijacking is a type of use-after-free attack in which the attacker utilizes dangling pointers in a program in order to abuse the control flow of it. Specifically, all virtual objects in C++ contain one or more pointers that point to the VTable. The VTable is a data structure that contains pointers to the method's implementation of the object. So, when a virtual object calls a method, the VTable pointer is used with an offset in order to be called the right method's implementation. When a virtual object is deallocated the memory space that the VTable object occupied is free for future allocations. Therefore, an attacker may intentionally spray the memory with data which are malicious pointers to a VTable under his control, and then just call a method through a dangling pointer and control the control flow of the process. VTPin secures binaries from such attacks by preserving any VTable pointer during the object's deallocation. Also, it pins the pointer to a safe VTable under his control, which provides safe methods to get be called if a dangling pointer tries to invoke a method. In this way, VTPin ensures the integrity of the control flow and provides help to log, track and patch possible dangling pointers. VTPin is a fully portable system which works directly on binaries. It does not require source compilation or binary rewriting, does not base on binary analysis and does not replace the allocator. Finally, it is fast and

with low memory overhead.

The main idea of VTPin is used for the implementation of ClassPin. Namely, we follow the same requirements and restrictions and we use the same logic to defeat against class pointer hijacking in Objective-C. Also, we used VTPin's technics for distinguishing virtual objects from other memory deallocations to implement ClassPin's algorithm that distinguishes `free` calls associated with objects from them which are not.

Chapter 9

Conclusion

In this thesis dissertation, we propose and implement a system for securing Objective-C software against class pointer hijacking through use-after-free vulnerabilities, *ClassPin*. ClassPin is a fully portable tool that protects directly binaries, without need of recompilation of the source code, by just preloading it along with the binary. Moreover, it does not base on complex binary analysis neither interferes with the allocator's policies and strategies. To succeed that, ClassPin interferes with each destruction of an object and preserves the class pointer. Finally, except protecting them from being hijacked, it also helps the developer track and patch possible dangling pointers. We implement two prototypes, one for Linux and one for MacOS platform, and we evaluate them by using Safari web browser on popular browser benchmarks suites. We show that ClassPin adds an average 0.38% -1.93% memory overhead and 21.12% - 28.43% runtime overhead, which could further be decreased as we explain. To sum up, the above characteristics make ClassPin suitable for practical and immediate application in software's protection.

Bibliography

[1] Advanced Exploitation of Mozilla Firefox Use-after-free Vulnerability (MFSA 2012-22). http://www.vupen.com/blog/20120625.Advanced_Exploitation_of_Mozilla_Firefox_UaF_CVE-2012-0469.php.

[2] Advanced Exploitation of Mozilla Firefox Use-After-Free Vulnerability (Pwn2Own 2014). http://www.vupen.com/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php.

[3] ARES-6 web browser benchmark. https://browserbench.org/ARES-6/about.html.

[4] Get information about resource utilization. http://www.manpages.info/macosx/getrusage.2.html.

[5] Hooking the memory allocator in Firefox. https://glandium.org/blog/?p=2848.

[6] How to Disable System Integrity Protection (SIP). https://support.intego.com/hc/en-us/articles/115003523252-How-to-Disable-System-Integrity-Protection-SIP-.

[7] IEEE Security and Privacy. https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=8013.

[8] JetStream web browser benchmark. https://browserbench.org/JetStream/.

[9] Mac OS DYLD. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/dyld.1.html.

[10] Memory allocation information. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man3/malloc_size.3.html.

[11] MotionMark web browser benchmark. https://browserbench.org/MotionMark/about.html.

[12] Objective-C Runtime. https://developer.apple.com/documentation/objectivec/objective_c_runtime.

[13] Octane web browser benchmark. https://developers.google.com/octane/benchmark.

[14] (Pwn2Own) Adobe Flash Player AS3 ConvolutionFilter Use-After-Free Remote Code Execution Vulnerability. http://www.zerodayinitiative.com/advisories/ZDI-15-134/.

[15] (Pwn2Own) Google Chrome Blink Use-After-Free Remote Code Execution Vulnerability. http://www.zerodayinitiative.com/advisories/ZDI-14-086/.

[16] realloc() – GNU C Library. http://bazaar.launchpad.net/~vcs-imports/glibc/master/view/head:/malloc/malloc.c#L4235.

[17] Resident set size. https://en.wikipedia.org/wiki/Resident_set_size.

[18] Save Current Environment (Non-local Gotos). http://man7.org/linux/man-pages/man3/setjmp.3.html.

[19] Speedometer web browser benchmark. https://browserbench.org/Speedometer2.0/.

[20] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proc. of ACM CCS*, pages 340–353, 2005.

[21] S. Andersen and V. Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[22] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proc. of ACM PLDI*, pages 157–164, 1991.

[23] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *Proc. of ISSTA*, pages 133–143, 2012.

[24] Chromium OS. Sandbox. https://www.chromium.org/developers/design-documents/sandbox.

[25] J. Evans. A Scalable Concurrent `malloc(3)` Implementation for FreeBSD. In *Proc. of BSDCan*, 2006.

[26] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proc. of USENIX SEC*, pages 475–490, 2012.

[27] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[28] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *Proc. of IEEE S&P*, pages 571–585, 2012.

[29] S. Lee, T. Johnson, and E. Raman. Feedback directed optimization of TCMalloc. In *Proc. of MSPC*, 2014.

[30] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. Subversive-c: Abusing and protecting dynamic message dispatch. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 209–221, Denver, CO, 2016. USENIX Association.

[31] G. McGraw. Software security. *IEEE Security Privacy*, 2(2):80–83, Mar 2004.

[32] Microsoft. Enhanced Mitigation Experience Toolkit, 2016. `http://www.microsoft.com/emet`.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proc. of IEEE S&P*, pages 601–615, 2012.

[34] PaX Team. Address Space Layout Randomization (ASLR), 2003. `http://pax.grsecurity.net/docs/aslr.txt`.

[35] V. Prevelakis and D. Spinellis. Sandboxing applications. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 119–126, Berkeley, CA, USA, 2001. USENIX Association.

[36] T. Rains, M. Miller, and D. Weston. Exploitation Trends: From Potential Risk to Actual Risk. In *RSA Conference*, 2015.

[37] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. Vtpin: Practical vtable hijacking protection for binaries. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 448–459, New York, NY, USA, 2016. ACM.

[38] A. Sotirov. Heap Feng Shui in JavaScript. In *Blackhat 2007*, 2007.

[39] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of ACM CCS*, pages 157–168, 2012.

Appendix A

In this appendix we show the ClassPin's source code for the MacOS prototype.

```
1
2  #define _GNU_SOURCE
3
4
5  #import <Foundation/Foundation.h>
6  #import <objc/runtime.h>
7  #import <objc/objc.h>
8
9
10 #import "safeObj.m"
11 #include <dlfcn.h>
12 #include <stdio.h>
13 #include <string.h>
14 #include <stdlib.h>
15 #include <signal.h>
16 #include <unistd.h>
17 #include <stdarg.h>
18 #include <sched.h>
19 #include <time.h>
20 #include <setjmp.h>
21
22
23 #include <sys/mman.h>
24
25 #include "MacOsMemoryMap.h"
26
27
28
29
30
31
32 /* Statistics  */
33 static unsigned long volatile stats_free_all = 0;
34
35 static unsigned long volatile stats_free_vobject = 0;
36 static unsigned long volatile stats_free_non_vobject = 0;
37 static unsigned long volatile stats_free_null = 0;
38 static unsigned long volatile stats_possible_vobject=0;
39 static  double stats_free_time=0;
40
41
42
43
44 static unsigned long volatile stats_total_size_of_vobjects = 0;
45 static unsigned long volatile stats_total_malloc_size = 0;
46
47 typedef void*(*real_malloc_t)(size_t);
48 static real_malloc_t real_malloc=NULL;
49
50
51 typedef void (*real_free_t)(void *ptr);
52 static real_free_t real_free = NULL;
```

```
53
54
55  typedef Class (*obj_setClass_t)(id *i, Class *cls);
56  static obj_setClass_t real_object_setClass = NULL;
57
58
59
60
61  static bool safeObjCreated = false;
62  static safeObj *safeobj = NULL;
63
64
65
66  /* Destructor:
67   Print stats when destroying vtpin object */
68  __attribute__((destructor)) static void print_vtpin_stats(void) {
69      struct rusage rusage;
70      getrusage(RUSAGE_SELF, &rusage);
71      size_t peakRSS=(size_t)rusage.ru_maxrss;
72
73      unsigned long stats_recorded = stats_free_non_vobject + stats_free_vobject +
                stats_free_null;
74      double stats_total_time=stats_free_time+stats_release_time;
75
76
77
78
79
80      printf( "\x1B[34m␣Objc-VTPin␣destroyed.␣Stats:\nfree()␣calls:␣%lu\nfree()␣calls␣
                recorded:␣%lu\nread_maps()␣calls:␣%lu\nfree()␣calls␣on␣objects:␣%lu\nfree()
                ␣calls␣on␣non␣objects:␣%lu\nfree()␣calls␣on␣null␣pointers:␣%lu\n\nfree()␣
                extra␣time␣%f␣seconds\nread_maps()␣extra␣time␣%f␣seconds\ntotal␣extra␣time␣
                %f␣seconds\nfree()␣total␣malloc_usable_size␣of␣Objects:␣%lu␣Bytes\nfree()␣
                total␣size␣of␣malloc'ed␣chunks:␣%lu␣Bytes\nfree()␣peak␣physical␣memory:␣%zu
                ␣Bytes\nObject␣overhead␣percentage(peakRSS):␣%g␣%%␣\nObject␣overhead␣
                percentage:␣%g␣%%\n--------------------------------------------\x1B[0m",
                stats_free_all,stats_recorded,stats_maps_all, stats_free_vobject,
                stats_free_non_vobject,stats_free_null, stats_free_time,stats_release_time,
                stats_total_time,stats_total_size_of_vobjects,stats_total_malloc_size,
                peakRSS,(((double)stats_total_size_of_vobjects)/peakRSS)*100,(((double)
                stats_total_size_of_vobjects)/stats_total_malloc_size)*100);
81      printf("\n");
82
83
84  }
85
86  static int segmeFlaf=0;
87
88  void handle_termination(int isignal, siginfo_t *pssiginfo, void *psContext) {
89      print_vtpin_stats();
90  }
91
92  sigjmp_buf point;
93
94  /* segmentation fault handling*/
95  void segfault_sigaction(int signal, siginfo_t *si, void *arg)
96  {
```

```
97      //return the execution to point
98      longjmp(point,1);
99  }
100
101
102
103
104 void establish_sighandler(void) {
105
106
107     struct sigaction newTERM;
108
109     memset(&newTERM, 0, sizeof newTERM);
110     sigemptyset(&newTERM.sa_mask);
111     newTERM.sa_sigaction = handle_termination;
112     newTERM.sa_flags = SA_SIGINFO;
113
114     int  res = sigaction(SIGTERM, &newTERM, NULL);
115     if (res) {
116         //    debug("Sigaction returned: %d", res);
117         exit(1);
118     }
119
120     struct sigaction sa;
121
122     memset(&sa, 0, sizeof(struct sigaction));
123     sigemptyset(&sa.sa_mask);
124     sa.sa_sigaction = segfault_sigaction;
125     sa.sa_flags   = SA_NODEFER;
126
127     sigaction(SIGSEGV, &sa, NULL);
128     sigaction(SIGBUS, &sa, NULL);
129
130 }
131
132
133
134
135 /* hook release*/
136 @interface NSObject (MHOverride)
137 -(oneway void)releaseMyedition;
138 @end
139
140 @implementation NSObject (MHOverride)
141 +(void)load{
142     Method method1= class_getInstanceMethod(self, @selector(release));
143     Method method2 = class_getInstanceMethod(self,@selector(releaseMyedition));
144
145     method_exchangeImplementations(method1,method2);
146 }
147
148 /*create safe object*/
149 static void initSafe(void) {
150     //NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
151     safeobj = [safeObj alloc];
152
153 }
```

```
154
155 /*release object by preserving and pinning class pointer*/
156 static void ClassPin_release_object(void *ptr) {
157     if (!safeObjCreated) {
158         initSafe();
159         safeObjCreated = true;
160     }
161
162
163     void *t;
164     t = (id)safeobj;
165
166     ptr = realloc(ptr, 8);
167     // printf("%d\n",malloc_usable_size(ptr));
168     memcpy(ptr, t, 8);
169     stats_free_vobject++;
170     stats_total_size_of_vobjects+=malloc_size(ptr);
171
172     // if(stats_free_vobject>stats_possible_vobject)
173     //     printf("%d %d\n",stats_free_vobject, stats_possible_vobject);
174
175 }
176
177
178
179
180 -(oneway void)releaseMyedition{
181     //  stats_release_all++;
182     //  struct timeval tim;
183     // gettimeofday(&tim,NULL);
184     // double start=tim.tv_sec+(tim.tv_usec/1000000.0);
185
186
187     if(!maps_loaded)
188         read_memory_maps();
189
190
191     //gettimeofday(&tim,NULL);
192     //double end=tim.tv_sec+(tim.tv_usec/1000000.0);
193     //stats_release_time+=(end-start);//difftime(end,start);
194     [self releaseMyedition];
195
196
197
198
199 }
200
201 @end
202
203 /*initialization of ClassPin*/
204 static void mymtrace_init(void) {
205
206
207
208     real_malloc = (real_malloc_t)dlsym(RTLD_NEXT, "malloc");
209     if (NULL == real_malloc) {
210         fprintf(stdout, "Error␣in␣`dlsym`:␣%s\n", dlerror());
```

```
211     }
212     //printf("initializing freeeee()\n");
213     real_free = (real_free_t)dlsym(RTLD_NEXT, "free");
214     //printf("%p\n",real_free);
215     if (NULL == real_free){
216         //fprintf(stdout, "error in free init\n");
217         exit(0);
218     }
219
220     establish_sighandler();
221
222 }
223
224 /* ---- DLOPEN START ---- */
225
226 typedef void *(*real_dlopen_t)(const char *, int);
227 static real_dlopen_t real_dlopen = NULL;
228 static bool newmaps=false;
229 void *dlopen(const char *filename, int flag) {
230     if (!real_dlopen) {
231         real_dlopen = (real_dlopen_t)dlsym(RTLD_NEXT, "dlopen");
232         if (!real_dlopen) {
233             printf("dlsym problem: %s", dlerror());
234             exit(1);
235         }
236     }
237
238     void *ret = real_dlopen(filename, flag);
239
240     if(maps_loaded)
241         newmaps=true;
242
243
244     return ret;
245 }
246
247 /*---- DLOPEN END ---- */
248
249
250 void *malloc(size_t size)
251 {
252     if(real_malloc==NULL) {
253         mymtrace_init();
254     }
255     stats_total_malloc_size+=size;
256     void *p = NULL;
257
258     // fprintf(stdout, "%d\tmalloc(%ld) = ",i++, size);
259     p = real_malloc(size);
260     //  fprintf(stdout, "%p\n", p);
261     return p;
262 }
263
264
265
266
267
```

```
268 static int inloop=0;
269 /*resolve if a pointer is associated with an object*/
270 static int checkPointer(void* ptr){
271
272     void *p;
273     char* x;
274
275     if(permissionsOf(ptr)!='w')
276         return 0;
277
278
279     if(inloop)
280         return 0;
281     if(setjmp(point)==0){
282         inloop=1;
283         //printf("%p\n",DEREFERENCE(ptr));
284         if(permissionsOf(GET_ADDRESS( DEREFERENCE(ptr)))!='w'&&permissionsOf(
                GET_ADDRESS( DEREFERENCE(ptr)))!='b'){
285             inloop=0;
286             return 0;
287         }
288
289         stats_possible_vobject++;
290         p=object_getClass(ptr);
291
292     }
293     else{
294         inloop=0;
295         return 0;
296     }
297
298     inloop=0;
299     if(p==NULL || p==nil||permissionsOf(p)!='w')
300         return 0;
301
302
303     if(permissionsOf(p)=='*'&&newmaps){
304
305         newmaps=false;
306         read_memory_maps();
307         // printf("maps loaded_again\n");
308         return checkPointer(ptr);
309     }
310
311
312
313
314     //printf("version - %d\n " ,class_getVersion(p));
315
316     return 1;
317 }
318
319 /*hook free*/
320 void free(void *ptr) {
321
322     //printf("ptr %p\n", ptr);
323     struct timeval tim;
```

```
324     gettimeofday(&tim,NULL);
325     double start=tim.tv_sec+(tim.tv_usec/1000000.0);
326
327     if (real_free == NULL)
328         mymtrace_init();
329     stats_free_all++;
330
331     if (ptr == NULL){
332         stats_free_null++;
333         return;
334     }
335     if (ptr == nil){
336         stats_free_null++;
337         return;
338     }
339
340
341     int flagVirtual=0;
342     if(maps_loaded){
343         flagVirtual= checkPointer(ptr);
344     }
345
346
347     if(flagVirtual){
348         ClassPin_release_object(ptr);
349         gettimeofday(&tim,NULL);
350         double end=tim.tv_sec+(tim.tv_usec/1000000.0);
351         stats_free_time+=(end-start);//difftime(end,start);
352     }else{
353         stats_free_non_vobject++;
354         gettimeofday(&tim,NULL);
355         double end=tim.tv_sec+(tim.tv_usec/1000000.0);
356         stats_free_time+=(end-start);//difftime(end,start);
357         real_free(ptr);
358     }
359
360
361
362     return;
363
364 }
```

Appendix B

In this appendix we show the ClassPin's source code for the Linux prototype.

```
1  #define _GNU_SOURCE
2
3  #import <Foundation/Foundation.h>
4  #import <objc/runtime.h>
5  #import <objc/objc.h>
6
7  #include "memoryMap.h"
8  #import "safeObj.m"
9  #include <dlfcn.h>
10 #include <stdio.h>
11 #include <string.h>
12
13 sigjmp_buf point;
14
15 /*handle segmentation fault*/
16 void segfault_sigaction(int signal, siginfo_t *si, void *arg)
17 {
18     //printf("Caught segfault at address %p\n", si->si_addr);
19     longjmp(point,1);
20 }
21
22 typedef void (*real_free_t)(void *ptr);
23 static real_free_t real_free = NULL;
24
25 typedef Class (*obj_setClass_t)(id *i, Class *cls);
26 static obj_setClass_t real_object_setClass = NULL;
27
28 /* Statistics  */
29 static unsigned long volatile stats_free_all = 0;
30 static unsigned long volatile stats_release_all=0;
31 static unsigned long volatile stats_free_vobject = 0;
32 static unsigned long volatile stats_free_non_vobject = 0;
33 static unsigned long volatile stats_free_null = 0;
34 static unsigned long volatile stats_possible_vobject=0;
35 static  double stats_free_time=0;
36 static double stats_release_time=0;
37
38 static bool safeObjCreated = false;
39 static safeObj *safeobj = NULL;
40 static void *code = (void *)0xfacedead;
41
42 /* Destructor:
43  Print stats when destroying vtpin object */
44 __attribute__((destructor)) static void print_vtpin_stats(void) {
45
46
47     unsigned long stats_recorded = stats_free_non_vobject + stats_free_vobject +
48     + stats_free_null;
49     double stats_total_time=stats_free_time+stats_release_time;
50
51
52
```

```
53
54
55      printf( "\x1B[34m␣Objc-VTPin␣destroyed.␣Stats:\nfree()␣calls:␣%lu\nfree()␣calls␣
                recorded:␣%lu\nobject_setClasss()␣calls:␣%lu\nfree()␣calls␣on␣vobjects:␣%lu
                \nobject_setClasss()␣calls␣on␣possible␣vobjects:␣%lu\nfree()␣calls␣on␣non␣
                vobjects:␣%lu\nfree()␣calls␣on␣null␣pointers:␣%lu\n\nfree()␣extra␣time␣%f␣
                seconds\nobject_setClasss()␣extra␣time␣%f␣seconds\ntotal␣extra␣time␣%f␣
                seconds␣%%\x1B[0m", stats_free_all,stats_recorded,stats_release_all,
                stats_free_vobject, stats_possible_vobject, stats_free_non_vobject,
                stats_free_null, stats_free_time,stats_release_time,stats_total_time);
56      printf("\n");
57
58
59  }
60
61  void handle_termination(int isignal, siginfo_t *pssiginfo, void *psContext) {
62      print_vtpin_stats();
63  }
64
65  void establish_sighandler(void) {
66
67      //    debug("Establishing sighandler");
68
69
70      struct sigaction newTERM;
71
72      memset(&newTERM, 0, sizeof newTERM);
73      sigemptyset(&newTERM.sa_mask);
74      newTERM.sa_sigaction = handle_termination;
75      newTERM.sa_flags = SA_SIGINFO;
76
77      int  res = sigaction(SIGTERM, &newTERM, NULL);
78      if (res) {
79          //    debug("Sigaction returned: %d", res);
80          exit(1);
81      }
82
83      struct sigaction sa;
84
85      memset(&sa, 0, sizeof(struct sigaction));
86      sigemptyset(&sa.sa_mask);
87      sa.sa_sigaction = segfault_sigaction;
88      sa.sa_flags   = SA_NODEFER;
89
90      sigaction(SIGSEGV, &sa, NULL);
91
92  }
93
94  static void initSafe(void) {
95      safeobj = [safeObj alloc];
96
97  }
98
99  static void release_object(void *ptr) {
100     if (!safeObjCreated) {
101         initSafe();
102         safeObjCreated = true;
```

```
103     }
104     void *t;
105     t = (id)safeobj;
106     // real_free(ptr);
107     ptr = realloc(ptr, 24);
108     // printf("%d\n",malloc_usable_size(ptr));
109     memcpy(ptr + 16, t, 8);
110     stats_free_vobject++;
111 }
112 
113 static void mtrace_init(void) {
114     real_free = (real_free_t)dlsym(RTLD_NEXT, "free");
115     if (NULL == real_free)
116         fprintf(stderr, "error in free init\n");
117 
118     real_object_setClass = (obj_setClass_t)dlsym(RTLD_NEXT, "object_setClass");
119     if (real_object_setClass == NULL) {
120         fprintf(stderr, "error in free init\n");
121         exit(0);
122     }
123     establish_sighandler();
124 
125 
126 }
127 
128 /* ---- DLOPEN START ---- */
129 
130 typedef void *(*real_dlopen_t)(const char *, int);
131 static real_dlopen_t real_dlopen = NULL;
132 
133 void *dlopen(const char *filename, int flag){
134     if (!real_dlopen) {
135         real_dlopen = (real_dlopen_t)dlsym(RTLD_NEXT, "dlopen");
136         if (!real_dlopen) {
137             debug("dlsym problem: %s", dlerror());
138             exit(1);
139         }
140     }
141 
142     void *ret = real_dlopen(filename, flag);
143     if(maps_loaded)
144         read_memory_maps();
145 
146 
147     return ret;
148 }
149 
150 /* ---- DLOPEN END ---- */
151 
152 
153 Class object_setClass(id object, Class cls) {
154     stats_release_all++;
155     struct timeval tim;
156     gettimeofday(&tim,NULL);
157     double start=tim.tv_sec+(tim.tv_usec/1000000.0);
158 
159 
```

```
160
161     if (real_object_setClass == NULL)
162         mtrace_init();
163
164
165     if (cls != (void *)0xdeadface){
166         gettimeofday(&tim,NULL);
167         double end=tim.tv_sec+(tim.tv_usec/1000000.0);
168         stats_release_time+=(end-start);//difftime(end,start);
169         return real_object_setClass((struct objc_class **)object, (struct objc_class
                **) cls);
170     }
171     stats_possible_vobject++;
172     gettimeofday(&tim,NULL);
173     double end=tim.tv_sec+(tim.tv_usec/1000000.0);
174     stats_release_time+=(end-start);//difftime(end,start);
175     return real_object_setClass((struct objc_class **)object, (struct objc_class **)
            object_getClass(object));
176 }
177
178
179 static int checkPointer(void* ptr){
180     void *p;
181     if(setjmp(point)==0)
182         p=object_getClass(ptr);
183     else
184         return 0;
185     if(p==NULL || p==nil)
186         return 0;
187
188     if(permissionsOf(p)[2]=='x')
189         return 0;
190
191     //printf("version - %d\n " ,class_getVersion(p));
192     //stats_possible_vobject++;
193     return 1;
194 }
195
196
197 void free(void *ptr) {
198     stats_free_all++;
199     struct timeval tim;
200     gettimeofday(&tim,NULL);
201     double start=tim.tv_sec+(tim.tv_usec/1000000.0);
202
203     if (real_free == NULL)
204         mtrace_init();
205
206     if (!maps_loaded)
207         read_memory_maps();
208
209     if (ptr == NULL){
210         stats_free_null++;
211         return;
212     }
213     if (ptr == nil){
214         stats_free_null++;
```

```
215             return;
216         }
217
218         // ptr is -16 from *object
219         if (checkPointer((ptr + 16)) == 1){
220
221             gettimeofday(&tim,NULL);
222             double end=tim.tv_sec+(tim.tv_usec/1000000.0);
223             stats_free_time+=(end-start);//difftime(end,start);
224             release_object(ptr);
225             //printf("free calls : %d\nvobjects : %d\n",freeCalled,vObjects);
226         }  else{
227             stats_free_non_vobject++;
228             gettimeofday(&tim,NULL);
229             double end=tim.tv_sec+(tim.tv_usec/1000000.0);
230             stats_free_time+=(end-start);//difftime(end,start);
231             real_free(ptr);
232         }
233         return;
234     }
```

Appendix C

In this appendix we show the source code for creating memory maps on MacOS prototype.

```
1
2  #include <stdio.h>
3  #include <string.h>
4
5  #define DEREFERENCE(o) (void *)(*(long *)o)
6  #define GET_ADDRESS(o) (void *)((long)o & 0x7fffffffffff) // 47bits
7  #define GET_ADDRESS2(o) (void *)((long)o & 0x7fffffff)   // 47bits
8  unsigned long long *startAddress;
9  unsigned long long *endAddress;
10 char *permissions;
11 char *readPerm;
12 static bool maps_loaded = false;
13 static unsigned long volatile stats_maps_all=0;
14 static double stats_release_time=0;
15 static int tableLength = 0;
16
17 static void read_memory_maps() {
18     stats_maps_all++;
19     struct timeval tim;
20     gettimeofday(&tim,NULL);
21     double start=tim.tv_sec+(tim.tv_usec/1000000.0);
22
23     int pid = getpid();
24     char *string = malloc(2000);
25     string[0] = '\0';
26     strcat(string, "sudo vmmap -interleaved -v ");
27
28     char *pidstr = malloc(10);
29     sprintf(pidstr, "%d", pid);
30     strcat(string, pidstr);
31     strcat(string, " | tail -n +23 > "tempFile.txt");
32 ____system(string);
33
34
35 ____FILE *fp = fopen(
36 ____________________"/tempFile.txt", "r");
37 ____//FILE *fp = fopen("tempfile.txt", "rb");
38 ____if (fp == NULL) {
39 ________printf("Error in opening file\n");
40 ________exit(0);
41 ____}
42
43 ____char c, prevC, prev2C;
44
45 ____// count lines of file
46 ____while ((c = getc(fp)) != EOF) {
47
48 ________if (prevC == '\n' && c == '\n')
49 ____________break;
50 ________if (c == '\n')
51 ____________tableLength++;
52 ________prevC = c;
```

```
53  ␣␣␣␣}
54
55  ␣␣␣␣//␣printf("%d——  lines ————————\n",tableLength);
56
57  ␣␣␣␣fseek(fp,␣0,␣SEEK_SET);
58  ␣␣␣␣if␣(maps_loaded)␣{
59  ␣␣␣␣␣␣␣␣free(startAddress);
60  ␣␣␣␣␣␣␣␣free(endAddress);
61  ␣␣␣␣␣␣␣␣free(permissions);
62  ␣␣␣␣␣␣␣␣free(readPerm);
63  ␣␣␣␣}
64  ␣␣␣␣startAddress␣=
65  ␣␣␣␣(unsigned␣long␣long␣*)malloc(sizeof(unsigned␣long␣long)␣*␣tableLength);
66  ␣␣␣␣endAddress␣=
67  ␣␣␣␣(unsigned␣long␣long␣*)malloc(sizeof(unsigned␣long␣long)␣*␣tableLength);
68  ␣␣␣␣permissions␣=␣(char␣*)malloc(tableLength);
69  ␣␣␣␣readPerm␣=␣(char␣*)malloc(tableLength);
70
71  ␣␣␣␣int␣i,␣k␣=␣0;
72  ␣␣␣␣char␣line[10000];
73  ␣␣␣␣char␣temp[20];
74  ␣␣␣␣int␣a␣=␣0;
75
76  ␣␣␣␣for␣(i␣=␣0;␣i␣<␣tableLength;␣i++)␣{
77  ␣␣␣␣␣␣␣␣while␣(1)␣{
78  ␣␣␣␣␣␣␣␣␣␣␣␣c␣=␣getc(fp);
79  ␣␣␣␣␣␣␣␣␣␣␣␣if␣(c␣!=␣'␣'␣&&␣prevC␣==␣'␣'␣&&␣prev2C␣==␣'␣')
80  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣break;
81  ␣␣␣␣␣␣␣␣␣␣␣␣prev2C␣=␣prevC;
82  ␣␣␣␣␣␣␣␣␣␣␣␣prevC␣=␣c;
83  ␣␣␣␣␣␣␣␣}
84  ␣␣␣␣␣␣␣␣prevC␣=␣'a';
85  ␣␣␣␣␣␣␣␣prev2C␣=␣'a';
86  ␣␣␣␣␣␣␣␣a␣=␣0;
87  ␣␣␣␣␣␣␣␣while␣(1)␣{
88  ␣␣␣␣␣␣␣␣␣␣␣␣c␣=␣getc(fp);
89  ␣␣␣␣␣␣␣␣␣␣␣␣if␣(c␣==␣'-')
90  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣break;
91  ␣␣␣␣␣␣␣␣␣␣␣␣temp[a++]␣=␣c;
92  ␣␣␣␣␣␣␣␣}
93  ␣␣␣␣␣␣␣␣temp[a]␣=␣'\0';
94  ␣␣␣␣␣␣␣␣//␣printf("%s ——  ",temp);
95  ␣␣␣␣␣␣␣␣startAddress[k]␣=␣strtol(temp,␣NULL,␣16);
96  ␣␣␣␣␣␣␣␣//␣if(k>0␣&&␣startAddress[k]!=endAddress[k-1])
97  ␣␣␣␣␣␣␣␣//␣␣startAddress[k]=endAddress[k-1];
98  ␣␣␣␣␣␣␣␣a␣=␣0;
99  ␣␣␣␣␣␣␣␣while␣(1)␣{
100 ␣␣␣␣␣␣␣␣␣␣␣␣c␣=␣getc(fp);
101 ␣␣␣␣␣␣␣␣␣␣␣␣if␣(c␣==␣'␣')
102 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣break;
103 ␣␣␣␣␣␣␣␣␣␣␣␣temp[a++]␣=␣c;
104 ␣␣␣␣␣␣␣␣}
105 ␣␣␣␣␣␣␣␣temp[a]␣=␣'\0';
106 ␣␣␣␣␣␣␣␣//␣printf("%s\n",temp);
107 ␣␣␣␣␣␣␣␣endAddress[k]␣=␣strtol(temp,␣NULL,␣16);
108
109 ␣␣␣␣␣␣␣␣while␣(1)␣{
```

```
110 ␣␣␣␣␣␣␣␣␣␣␣␣c␣=␣getc(fp);
111 ␣␣␣␣␣␣␣␣␣␣␣␣if␣(c␣==␣']')
112 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣break;
113 ␣␣␣␣␣␣␣␣}
114 ␣␣␣␣␣␣␣␣permissions[k]␣=␣'n';␣//␣none
115 ␣␣␣␣␣␣␣␣c␣=␣getc(fp);
116 ␣␣␣␣␣␣␣␣c␣=␣getc(fp);
117 ␣␣␣␣␣␣␣␣readPerm[k]␣=␣c;
118 ␣␣␣␣␣␣␣␣c␣=␣getc(fp);
119 ␣␣␣␣␣␣␣␣if␣(c␣==␣'w')␣{
120 ␣␣␣␣␣␣␣␣␣␣␣␣permissions[k]␣=␣'w';
121 ␣␣␣␣␣␣␣␣}
122 ␣␣␣␣␣␣␣␣c␣=␣getc(fp);
123 ␣␣␣␣␣␣␣␣if␣(c␣==␣'x')␣{
124 ␣␣␣␣␣␣␣␣␣␣␣␣if␣(permissions[k]␣==␣'w')
125 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣permissions[k]␣=␣'b';␣//␣both
126 ␣␣␣␣␣␣␣␣␣␣␣␣else
127 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣permissions[k]␣=␣'x';
128 ␣␣␣␣␣␣␣␣}
129
130 ␣␣␣␣␣␣␣␣k++;
131
132 ␣␣␣␣␣␣␣␣while␣(1)␣{
133 ␣␣␣␣␣␣␣␣␣␣␣␣c␣=␣getc(fp);
134 ␣␣␣␣␣␣␣␣␣␣␣␣if␣(c␣==␣'\n')
135 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣break;
136 ␣␣␣␣␣␣␣␣}
137 ␣␣␣␣␣␣␣␣//
138 ␣␣␣␣}
139
140 ␣␣␣␣fclose(fp);
141 ␣␣␣␣free(string);
142 ␣␣␣␣system("rm /tempFile.txt");
143 ␣␣␣␣maps_loaded␣=␣true;
144
145 ␣␣␣␣gettimeofday(&tim,NULL);
146 ␣␣␣␣double␣end=tim.tv_sec+(tim.tv_usec/1000000.0);
147 ␣␣␣␣stats_release_time+=(end-start);//difftime(end,start);
148
149
150 ␣␣␣␣return;
151 }
152 //␣static␣int␣a=0;
153 //␣static␣int␣b=0;
154 static␣char␣permissionsOf(void␣*ptr)␣{
155 ␣␣␣␣//␣printf("%p\n",ptr);
156 ␣␣␣␣if␣(ptr␣==␣NULL␣||␣ptr␣==␣nil␣||␣!ptr)
157 ␣␣␣␣␣␣␣␣return␣'*';
158 ␣␣␣␣//␣b++;
159 ␣␣␣␣//␣void␣*ptr␣=␣DEREFERENCE(p);
160 ␣␣␣␣//␣printf("is read only %p %p \n",␣p,␣ptr);
161 ␣␣␣␣int␣i␣=␣0;
162 ␣␣␣␣for␣(i␣=␣0;␣i␣<␣tableLength␣-␣2;␣i++)␣{
163 ␣␣␣␣␣␣␣␣if␣((unsigned␣long␣long)ptr␣>=␣startAddress[i]␣&&
164 ␣␣␣␣␣␣␣␣␣␣␣␣(unsigned␣long␣long)ptr␣<=␣endAddress[i])
165 ␣␣␣␣␣␣␣␣␣␣␣␣break;
166 ␣␣␣␣␣␣␣␣else␣if␣(i␣==␣tableLength␣-␣2)
```

```
167 ␣␣␣␣␣␣␣␣␣␣␣return␣'*';
168 ␣␣␣␣}
169 ␣␣␣␣//␣printf("%x  %x  %c\n",startAddress[i],endAddress[i],permissions[i]);
170 ␣␣␣␣//␣sleep(5);
171 ␣␣␣␣return␣permissions[i];
172 }
173
174 static␣int␣readPermissionsOf(void␣*ptr)␣{
175 ␣␣␣␣//␣b++;
176 ␣␣␣␣//␣void␣*ptr␣=␣DEREFERENCE(p);
177 ␣␣␣␣//␣printf("is read only %p  %p \n",␣p,␣ptr);
178 ␣␣␣␣int␣i␣=␣0;
179 ␣␣␣␣for␣(i␣=␣0;␣i␣<␣tableLength␣-␣2;␣i++)␣{
180 ␣␣␣␣␣␣␣␣if␣((unsigned␣long␣long)ptr␣>=␣startAddress[i]␣&&
181 ␣␣␣␣␣␣␣␣␣␣␣␣(unsigned␣long␣long)ptr␣<=␣endAddress[i])
182 ␣␣␣␣␣␣␣␣␣␣␣␣break;
183 ␣␣␣␣␣␣␣␣else␣if␣(i␣==␣tableLength␣-␣2)
184 ␣␣␣␣␣␣␣␣␣␣␣␣return␣'*';
185 ␣␣␣␣}
186 ␣␣␣␣//␣printf("%x  %x  %s\n",startAddress[i],endAddress[i],permissions[i]);
187 ␣␣␣␣//␣sleep(5);
188 ␣␣␣␣if␣(readPerm[i]␣==␣'r')
189 ␣␣␣␣␣␣␣␣return␣1;
190 ␣␣␣␣else
191 ␣␣␣␣␣␣␣␣return␣0;
192 }
```

Appendix D

In this appendix we show the source code for creating memory maps in Linux prototype.

```
1
2  #include <stdio.h>
3  #include <string.h>
4
5  #define DEREFERENCE(o) (void *)(*(uintptr_t *)o)
6
7  unsigned long long *startAddress;
8  unsigned long long *endAddress;
9  char **permissions;
10 static bool maps_loaded = false;
11 static int tableLength = 0;
12
13 static void read_memory_maps() {
14     maps_loaded = true;
15     // NSProcessInfo *processInfo = [NSProcessInfo processInfo];
16     // int processID = [processInfo processIdentifier];
17     int processID = getpid();
18
19
20     char *mapFilename = malloc(16);
21     mapFilename[0] = '\0';
22     sprintf(mapFilename, "/proc/%d/maps", processID);
23     // printf("%s\n",x);
24     FILE *fp = fopen(mapFilename, "rb");
25     if (fp == NULL) {
26         printf("Error in opening %s\n", mapFilename);
27         exit(-1);
28     }
29
30     char c;
31
32     // count lines of file
33     while ((c = getc(fp)) != EOF) {
34         if (c == '\n')
35             tableLength++;
36     }
37     fseek(fp, 0, SEEK_SET);
38
39     startAddress =
40     (unsigned long long *)malloc(sizeof(unsigned long long) * tableLength);
41     endAddress =
42     (unsigned long long *)malloc(sizeof(unsigned long long) * tableLength);
43     permissions = (char **)malloc(sizeof(char *) * tableLength);
44
45     int k = 0;
46     char line[10000];
47     char temp[20];
48     int a = 0;
49
50     while ((c = getc(fp)) != EOF) {
51         if (c != '\n') {
52             line[a++] = c;
```

```
53              continue;
54          }
55
56          a = 0;
57          while (line[a] != '-') {
58              temp[a] = line[a];
59              a++;
60          }
61
62          startAddress[k] = strtol(temp, NULL, 16);
63          a++;
64
65          int q = 0;
66          while (line[a] != ' ') {
67              temp[q] = line[a];
68              q++;
69              temp[q] = '\0';
70              a++;
71          }
72
73          endAddress[k] = strtol(temp, NULL, 16);
74
75          while (line[a++] != ' ')
76              ;
77          // a++;
78          permissions[k] = (char *)calloc(5, sizeof(char));
79          strncat(permissions[k], line + a, 4);
80          k++;
81          a = 0;
82      }
83
84  }
85
86  static char *permissionsOf(void *ptr) {
87
88      int i = 0;
89      for (i = 0; i < tableLength - 2; i++) {
90          if ((unsigned long long)ptr >= startAddress[i] &&
91              (unsigned long long)ptr <= endAddress[i])
92              break;
93      }
94
95      return permissions[i];
96  }
```

Appendix E

In this appendix we show the source code of the ClassPin's safe class. This code is the same for the two prototypes.

```
1
2  #import <Foundation/Foundation.h>
3
4  @interface safeObj : NSObject
5  //+ (void) createSafeObject;
6  + (void)safe;
7  @end
8
9  @implementation safeObj
10 /*forward any method call*/
11 - (void)forwardInvocation:(NSInvocation *)inv {
12     printf("\033[1m\033[31m""
               ************************************************************\n**A␣
               reserved␣pointer␣at␣%p␣called␣method␣:␣%s\n
               ************************************************************""\033[0m",
               self ,sel_getName([inv selector]));
13     printf("\n");
14
15
16 }
17
18 - (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector {
19     int numArgs = [[NSStringFromSelector(aSelector)
20                     componentsSeparatedByString:@":"] count] -
21     1;
22     // we assume that all arguments are objects
23     // The type encoding is "v@:@@@...", where "v" is the return type, void
24     // "@" is the receiver, object type; ":" is the selector of the current
25     // method;
26     // and each "@" after corresponds to an object argument
27     return [NSMethodSignature
28             signatureWithObjCTypes:[[@"v@:" stringByPaddingToLength:numArgs + 3
29                                                          withString:@"@"
30                                                     startingAtIndex:0] UTF8String]];
31 }
32
33
34 + (void)safe {
35     printf("Safe␣objected␣called!!\n");
36 }
37
38 @end
```