POP and PUSH: Demystifying and Defending against (Mach) Port-Oriented Programming

Min Zheng*, Xiaolong Bai*, Yajin Zhou†⋆ , Chao Zhang‡§ and Fuping Qu*

* Orion Security Lab, Alibaba Group
† Zhejiang University
‡ Institute for Network Science and Cyberspace, Tsinghua University
§ Beijing National Research Center for Information Science and Technology

***Abstract*—Apple devices (e.g., iPhone, MacBook, iPad, and Apple Watch) are high value targets for attackers. Although these devices use different operating systems (e.g., iOS, macOS, iPadOS, watchOS, and tvOS), they are all based on a hybrid kernel called XNU. Existing attacks demonstrated that vulnerabilities in XNU could be exploited to escalate privileges and jailbreak devices. To mitigate these threats, multiple security mechanisms have been deployed in latest systems.**

In this paper, we first perform a systematic assessment of deployed mitigations by Apple, and demonstrate that most of them can be bypassed through corrupting a special type of kernel objects, i.e., Mach port objects. We summarize this type of attack as *(Mach) Port Object-Oriented Programming* (POP). Accordingly, we define multiple attack primitives to launch the attack and demonstrate realistic scenarios to achieve full memory manipulation on recently released systems (i.e., iOS 13 and macOS 10.15). To defend against POP, we propose the *Port Ultra-SHield* (PUSH) system to reduce the number of unprotected Mach port objects. Specifically, PUSH automatically locates potential POP primitives and instruments related system calls to enforce the integrity of Mach port kernel objects. It does not require system modifications and only introduces 2% runtime overhead. The PUSH framework has been deployed on more than *40,000* macOS devices in a leading company. The evaluation of 18 public exploits and one zero-day exploit detected by our system demonstrated the effectiveness of PUSH. We believe that the proposed framework will facilitate the design and implementation of a more secure XNU kernel.

I. Introduction

XNU-based operating systems, e.g., iOS and macOS, are popular targets for attackers, possibly due to potential financial gains [52]. Attackers could exploit vulnerabilities to escalate the privilege to a higher one, i.e., the root privilege. However, the root user of iOS and macOS cannot access or modify important system files. Thus, in order to get *full* control of the system (e.g., jailbreaking the system), attackers have to find a way to bypass this limitation.

Memory corruption *in the operating system kernel* is a common way to achieve this goal. For instance, buffer overflow and use-after-free (UAF) are two types of vulnerabilities that could cause memory corruption. With a buffer overflow vulnerability, return addresses on the stack or kernel objects in the heap can be corrupted. In the case of a UAF vulnerability, a previously freed object with a function pointer could be abused by attackers to hijack the control flow.

Furthermore, if attackers know addresses of some useful code gadgets, e.g., via arbitrary kernel *read/write primitives*, they can leverage code-reuse attacks, including return-oriented programming (ROP) [64] or return-to-libc [3], to redirect the control flow to these gadgets and gain the capability of *kernel code execution*.

With the development of vulnerability detection technique, e.g., the fuzz testing [5, 54, 66], the number of memory corruption vulnerabilities discovered in XNU has increased rapidly in recent years [8]. Accordingly, to raise the bar for attackers and prevent the vulnerabilities from being exploited, several mitigations [20], e.g., DEP [12], ASLR [2] and CFI [53], have been adopted by the latest iOS and macOS systems.

However, attackers have abused a special type of kernel object, i.e., Mach port, to bypass these mitigations [33, 36, 51]. In XNU, a Mach port is represented by a pointer to an `ipc_port` object, which is used to represent abstractions (e.g., task and thread) in the kernel. User space applications could interact with Mach port objects to access the underlying abstractions by sending Mach messages to ports. In order to invoke a system functionality that is represented by a Mach Port, an application first asks the kernel for accessing a port. Then it leverages the IPC (Inter Process Communication) mechanism to send Mach messages to that port. By corrupting this kernel object, all existing defense mechanisms adopted in the recently released XNU systems (i.e., iOS 13 and macOS 10.15) can be bypassed. By doing so, an attacker can achieve multiple primitives (e.g., kernel memory read/write) by simply issuing system calls in user space. The root cause is that Mach port objects are unconditionally trusted by the kernel. Indeed, the Mach port object has been exploited in several known attacks. However, how to *systematically* leverage the Mach port objects to gain steady kernel memory read, write and execution capabilities is still unknown by the community.

In this paper, we first review the mitigation techniques in recent released XNU systems (i.e., iOS 13 and macOS 10.15) and illustrate how they make the traditional exploitation techniques ineffective. Based on public exploits, we then systematically summarize *(Mach) Port-Oriented Programming* (POP), an attack technique that leverages Mach port kernel objects to bypass these mitigations. Specifically, because the kernel unconditionally trusts Mach port objects without effec-

⋆Corresponding author.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23126
www.ndss-symposium.org

tive verification, if the attacker has the capability to corrupt kernel port objects, he or she could achieve multiple primitives through user space system calls.

We then propose the *Port Ultra-SHield* (PUSH) to defend against this attack. PUSH automatically locates unsafe Mach port objects based on the primitive models and instruments related system calls to deploy security policies. Specifically, it constructs Mach port-oriented control flow (POC) graphs and locates Mach port objects that could potentially be corrupted and abused to launch the POP attack. Then, PUSH uses examiners to check the integrity of these kernel objects (i.e., objects that can potentially be corrupted by attackers) before or after specified system calls. To make our solution practical, PUSH is deployed through kernel extensions with negligible overheads, without requiring any modification to the XNU kernel.

We evaluated PUSH with 18 public exploits. All of them could be detected and blocked by our system. This demonstrated the effectiveness of our system with small performance overhead (2%). In addition, the PUSH framework has been deployed on more than 40,000 MacBooks in Alibaba Group [4]. It successfully detected mutated public exploits and a previously unknown exploit. We have discussed POP and PUSH system with Apple's security team (follow-up id: 707542859). They appreciate our efforts to improve the security of their products. Accordingly, security enhancements like object address verification (e.g., `zone_require()` [50]) and object data verification (e.g., `data-PAC` [9]) for Mach port objects have been deployed on later released XNU systems (e.g., iOS 14 beta).

In summary, this paper makes the following main contributions:

- We summarize POP, a (Mach) port-oriented attack technique that can bypass most of XNU kernel mitigations, and demonstrate the way to achieve multiple attack primitives (Section III).
- We present POP primitive searcher, a static detection tool to locate exploitable Mach port related objects and system calls (Section IV).
- We propose PUSH, a generic kernel object protection framework for XNU. It enforces kernel object integrity to prevent the corruption of Mach port objects and defend against the (Mach) port-oriented programming attack (Section V).
- We show the effectiveness of PUSH through the detection and blocking of 18 public exploits. PUSH does not require system modifications and only introduces 2% runtime overhead (Section VI). Moreover, the PUSH framework has been deployed in a leading company with more than *40,000* macOS devices and successfully detected new exploits (Section VII).

II. Background

In this section, we will briefly present the necessary background knowledge to understand our proposed system.

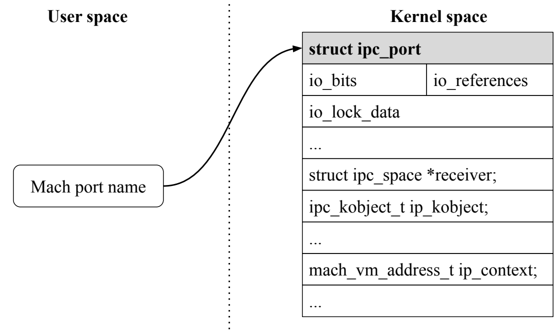

Fig. 1: Mach port name and the `ipc_port` structure [13].

A. Mach port

A Mach port in XNU is a kernel controlled communication channel between user-space processes and the kernel. A process with appropriate privilege is able to send messages to a port. Ports are used to represent resources, services, and facilities (e.g., hosts, tasks, threads, memory objects, and clocks). It provides object-style access to these abstractions. To access resources in the kernel, a process could send Mach messages to a specific type of the Mach port. In user space, a Mach port name is an integer number representing a handler for a Mach port in the kernel space. In kernel, a Mach port is represented by a pointer to an `ipc_port` object.

Figure 1 shows the structure of `ipc_port`. There are 40 types of `ipc_port` objects in XNU-6153.11.26 (the kernel of iOS 13 and macOS 10.15). The `io_bits` field defines the type of the object. For instance, an `ipc_port` object with the `IKOT_TASK` type represents a Mach task, which is a machine-independent abstraction of the execution environment of threads. The `io_references` field counts the reference number of the object. If the number decreases to zero, the kernel object will be freed. Locking related data is stored in the `io_lock_data` field. The `receiver` field is a pointer that points to the receiver's IPC space (e.g. `ipc_space_kernel`). The `ip_kobject` field points to a kernel data structure according to the kernel object type. Note that there is no type integrity check for `ipc_port` objects. Hence, the attacker can change the type (`io_bits` field) of the object through a memory corruption vulnerability, leading to a type-confusion attack.

B. Kernel zone & heap feng shui

In order to manage the heap memory available in the kernel space, the XNU kernel uses the zone allocator. It is similar to the slab in Linux and the pool in Windows. In particular, a zone is a collection of fixed-size memory blocks (a.k.a, elements). They are accessible from efficient interfaces for memory allocation and deallocation. Specifically, the `kalloc()` function family contains the most frequently used memory allocation functions. They provide the access to a fast general-purpose memory allocator built on top of the zone allocator. The `kalloc()` function supports a set of allocation sizes, ranging from 16 bytes to several kilobytes (each size is a power of 2). When the allocator is initialized, it creates a zone for each allocation size and the name of the zone reflects its size. For instance, the zone name `kalloc.32` means that the zone's size is 32 bytes.

During the exploitation, attackers need to manipulate the heap memory layout to ensure that a particular kernel object will most likely reside inside a particular zone. This technique is called heap feng shui [17]. For instance, in order to overwrite the function pointer inside a kernel object (the target kernel object), attackers can spray lots of kernel objects on the heap, which have the same size with the target object. Since the newly created objects are likely adjacent to the target kernel object, attackers can corrupt the interested kernel object in a predictable way.

In earlier versions of XNU, the `freelist` contains all the freed kernel objects inside a zone with a LIFO (last-in-first-out) policy. Hence, the layout of kernel objects in the same size zone is predictable. To make the object layout hard to be predicted, Apple deployed a mitigation called `random_free_to_zone` in iOS 9.2 and macOS 10.11.2. When a new zone block is allocated, XNU will randomly choose the first or last position in the block and add it into the `free_elements` list. In this case, it is hard for attackers to corrupt the kernel objects in a predictable way. In Section IV-A, we will introduce a type of primitive (querying primitive) in the POP attack to bypass the zone element randomization.

C. Mitigations

The arm-race between Apple and the jailbreaking community (and attackers) is like an escalating game of cat and mouse. In particular, to raise the bar for successful exploitation of vulnerabilities, Apple has developed corresponding defenses, e.g., many hardware-assisted protections including KIP [25] for iOS devices. However, they are mainly used to prevent attackers from patching the kernel (after gaining the capability of kernel memory manipulation). Instead, POP escalates an attacker's capability from a limited memory corruption to arbitrary kernel memory manipulation. In the following, we will illustrate deployed mitigations that are related to the POP attack.

DEP/KASLR Apple has deployed Data Execution Prevention (DEP) and Kernel Address Space Layout Randomization (KASLR) since iOS 6 and macOS 10.8. DEP is a system-level memory protection feature. It ensures that a memory page cannot be writable and executable at the same time to prevent the code injection attack. To break DEP, the code-reuse attack (e.g., ROP) was proposed. It utilizes the code that is already present in the memory (e.g., ROP gadgets). In particular, after hijacking the control flow, attackers could redirect the control flow to execute ROP gadgets. This attack requires the knowledge of gadget addresses in the memory. KASLR randomizes the locations of various memory segments (e.g., code and data) to make the gadget addresses hard to be predicted. To bypass KASLR, attackers usually leverage information leakage vulnerabilities or corrupted kernel objects to obtain addresses of gadgets in the memory.

CFI Control-flow integrity (CFI) is another type of the defense mechanism. It ensures that runtime control transfers follow a pre-computed control flow graph that consists of valid control flow transfers. A hardware-assisted (Pointer Authentication Codes) CFI is deployed on Apple devices (iPhone 11 and iPhone XS) with the A12/A13 chip. However, attackers can still bypass the protection to escalate privileges through POP attacks (case study 2 and 3 in Section VI-A).

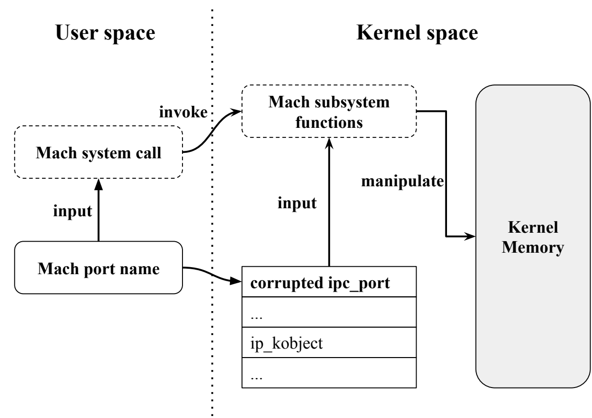

Fig. 2: An example flow of the POP attack. It first corrupts an `ipc_port` kernel object. After that, kernel functions will be invoked through a system call to achieve various attack primitives, e.g., manipulating the kernel memory (as shown in the graph). Other primitives could also be used during a real attack.

D. The Traditional Attack Flow

Traditional attacks to the XNU kernel usually request both an information leak vulnerability and a control flow hijacking vulnerability. The information leak vulnerability is used to break KASLR. By doing so, an attacker could calculate gadget addresses and construct a ROP chain. With the control-flow hijacking vulnerability, an attacker could redirect the control flow to execute the ROP gadgets. Depending on the found ROP gadgets, different primitives like arbitrary kernel memory read and write can be achieved. However, with the deployment of CFI, hijacking the control flow becomes hard. Thus, POP, the (Mach) port-oriented attack is proposed.

III. Demystifying (Mach) Port-oriented Programming

The Mach subsystem in the XNU kernel is responsible for receiving Mach messages and performing requested operations on resources such as tasks and threads. Consequently, by issuing Mach system calls and counterfeiting `ipc_port` objects, attackers can achieve useful attack primitives. In this section, we summarize a kind of attacks [33, 36, 51] that leverage Mach ports to achieve arbitrary kernel memory manipulation. The attack is launched through issuing multiple system calls. Since it is mainly based on the Mach port kernel object, we call it *(Mach) Port-oriented Programming* (`POP`) in the paper. Figure 2 illustrates the flow of a POP attack.

A. Assumptions

The main goal of the POP attack is to manipulate kernel memory content and layout, even in the case that multiple mitigations discussed in Section II-C are present in the system. Our attack assumes an existence of a memory corruption vulnerability that could be used to corrupt a kernel object. This is a fair assumption since such vulnerabilities are common nowadays [22, 31, 33, 46]. POP escalates an attacker's capability from a limited memory corruption to arbitrary kernel memory manipulation. After that, further attacks could be

launched, such as directly changing critical kernel data, e.g., conditional variables or function pointers.

B. POP Definition

The POP attack is based on counterfeiting the `ipc_port` kernel object. It consists of multiple attack primitives and can defeat existing defense mechanisms.

To help readers better understand POP, we classify primitives used in the POP attack according to their behavior:

Querying Primitive (QP) `QP` is used to break the randomization-based mitigation and help the attacker to find the handler of the corrupted port in kernel without crashing the system. However, the attacker can only gain limited information (e.g., `error` return value) from `QP`. A successful attack usually requires multiple invocations of this primitive.

Kernel Memory Read Primitive (RP) `RP` helps the attacker to gain partial or arbitrary kernel memory read capability. It abuses corrupted port objects with an inadequate check of the type integrity. The limitation is it can be invalidated by adding type integrity checks. Besides, it is usually hard to find alternative ones.

Kernel Memory Write Primitive (WP) `WP` helps the attacker to change the memory in limited or arbitrary kernel addresses. In general, XNU provides lots of ways to legitimately change the data in the kernel. But it is hard for attackers to write arbitrary data to arbitrary addresses. A powerful `WP` usually leverages a fake `ipc_port` object that consists of additional kernel information (e.g., the `map` memory space of `kernel_task`) obtained through `RP`.

Privileged Purpose Primitive (PPP) XNU provides powerful system calls for privileged ports (e.g., `host_priv` and `kernel_task`). If an attack can obtain a send right to privileged ports, he or she can enlarge the attack surface or even control the whole system. We summarize the abuses of these Mach system calls as privileged purpose primitives.

Other Attack Primitives After obtaining arbitrary kernel memory manipulation capabilities, the attacker has multiple strategies for further attacks. In this section, we use the arbitrary code execution primitive (`ACEP`) as an example. If the kernel CFI is not enforced, the attack can achieve this by corrupting function pointers or `vtable` (the traditional way). If the CFI is deployed, it can be bypassed through a key forgery strategy [14]. It's worth noting that this primitive is not necessary for POP to be effective. We can leverage previously discussed ones (e.g., `PPP` and/or `WP`) to gain privilege escalation through directly changing critical kernel data.

Note that most POP primitives are only changing non-control data. For `QP`, `RP`, `WP`, and `PPP`, the counterfeit objects only contain fake data or pointers to other objects. The function pointers are not changed. However, for `ACEP`, the attacker needs to corrupt function pointers inside the object.

C. POP Attack Chain

A POP attack chain consists of a sequence of POP primitives: from a weak primitive such as a limited memory corruption vulnerability to a strong primitive, e.g., arbitrary kernel memory manipulation. In general, a successful POP attack consists of the following steps.

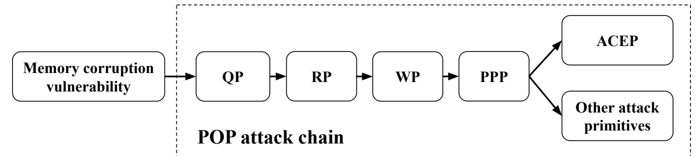

Fig. 3: The overview of the POP attack chain and dependencies between primitives. An arrow from A to B means the primitive B depends on the primitive A.

- Step 1: The attacker leverages a memory corruption vulnerability to gain a send right to a fake `ipc_port` object whose fields can be read and written by the attacker.
- Step 2: The `QP` primitive helps the attacker to find the right port name (i.e., the handler of the fake `ipc_port` object) in user space and provides enough kernel object information for the next primitive (`RP`).
- Step 3: After getting the handler of the fake `ipc_port` object, the attacker can use `RP` to read kernel memory by modifying the content of the fake `ipc_port` object.
- Step 4: With enough kernel information (e.g., the `map` memory space of the `kernel_task` and the `ipc_space_kernel` receiver of kernel space) obtained through `RP`, the attacker can construct a fake kernel task port object to get an arbitrary kernel memory write ability through `WP`.
- Step 5: The attacker can modify the critical kernel data through `WP` to get send rights to privileged ports in user space. Through privileged ports, the attacker can leverage `PPP` to achieve privileged operations (e.g., arbitrary process and thread manipulation) and other primitives. For instance, arbitrary code execution primitive (`ACEP`) can be implemented by changing the function pointers of kernel objects via `PPP`.

Figure 3 shows the overview of a POP attack chain and dependencies between primitives. For instance, `RP` depends on the `QP` to bypass KASLR and provide the information of kernel objects. The information is needed to construct a fake `ipc_port` object. Besides, traditional exploit technique needs arbitrary code execution primitives to perform memory read/write through ROP. In contrast, POP attack gains kernel memory read and write capability through standard system calls (`RP` and `WP`) with counterfeit kernel objects. These primitives are abstractions of attacking capabilities. They exist in multiple types of port objects that can be reached by user programs through Mach messages.

D. An Example of the POP Attack: Yalu Exploit

Yalu [51] is a POP based exploit that can be leveraged to jailbreak iOS 10.2. In the following, we illustrate each step of this exploit and POP primitives used.

Making a fake Mach port via a heap overflow vulnerability CVE-2017-2370 is a heap overflow vulnerability in the `mach_voucher_extract_attr_recipe_trap()` function. Yalu exploit first sends lots of Mach messages with

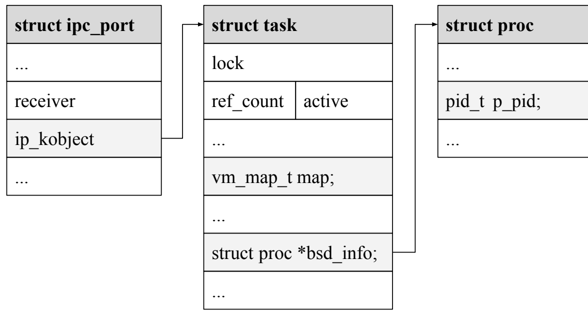

Fig. 4: `pid_for_task()` and `mach_vm_*()` related structures [13].

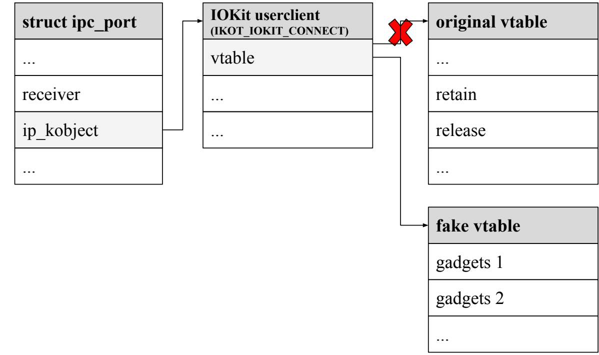

Fig. 5: Modifying an IOKit userclient's vtable [13].

the OOL (out-of-line) port that points to the kernel memory. It then uses the heap overflow vulnerability to overflow an OOL port pointer to make it point to a fake `ipc_port` object in user space. After receiving the Mach messages, Yalu exploit gains a valid send right to the fake `ipc_port` object whose fields can be read and written by the attacker.

Breaking KASLR via QP With KASLR, base addresses of kernel components are randomized. But some global kernel objects (e.g., Mach clock) are stored in a fixed place in the kernel. For instance, the function `clock_sleep_trap()` (Listing 7 in Appendix) is a system call expecting its first argument to be a send right to the global system clock. This function will return `KERN_SUCCESS` if the port name is correct. Therefore, this system call could be converted into a `QP`. Consequently, Yalu exploit counterfeits the `kobject` field of `ipc_port` object with `IKOT_CLOCK` type and invokes the `clock_sleep_trap()` system call to lunch a brute force attack to guess the address of the global system clock and break KASLR.

Getting arbitrary kernel memory read capability via RP By using the type-confusion attack, `RP` can leverage some system calls to copy sensitive data between the kernel space and the user space. For instance, `pid_for_task()` is such a system call (Listing 8 in Appendix) which returns the PID number corresponding to a particular Mach task. However, this function does not check the validity of the `task`, and directly returns the value of `task->bsd_info->p_pid` to the user space after calling `get_bsdtask_info()` and `proc_pid()` (Figure 4). Therefore, by carefully adjusting the offsets in a fake task structure, it is possible to convert this system call into a `RP`. Consequently, Yalu exploit leverages the `pid_for_task()` system call to gain an arbitrary kernel memory read primitive via a fake `IKOT_TASK` object.

Manipulating the kernel memory via PPP XNU provides a powerful set of routines such as `mach_vm_*()` for user space programs to manipulate their own memory. In general, a process only has the privilege to access its own task port and memory. However, Yalu exploit uses `RP` to create a fake kernel task to bypass the ownership check of the memory space. Then it leverages `PPP`s (i.e., `mach_vm_*()`) to manipulate the kernel memory.

Executing arbitrary code in kernel via ACEP I/OKit is an object-oriented device driver framework. Its object contains a virtual table. Because of `PPP`, Yalu exploit can manipulate the kernel memory. Therefore, To gain a `ACEP`, Yalu exploit changes the vtable entry of an I/OKit object and invokes `iokit_user_client_trap()` to redirect the control flow to the address of a ROP gadget (Figure 5). With `PPP`s and `ACEP`s, Yalu exploit patches the kernel functions and finally jailbreaks the device. Note that this step requires the change of a function pointer. Thus, the deployment of CFI can make this step ineffective. However, other methods [14] to change critical kernel data could be used.

IV. POP PRIMITIVE SEARCHER

In this section, we propose our POP primitive searcher, a static analysis tool to locate exploitable Mach port related objects and potential POP primitives. On the one hand, an attacker can leverage this tool to facilitate a POP attack. On the other hand, it can be used to help defend against the POP attack (Section V).

A. Primitive Model

To better describe a POP functionality, we use the following structure (called Primitive Model) to define the primitive. In particular, the primitive model is used to define a possible attacking behavior (primitive) based on the code sequence with specific kernel objects and code patterns. The syntax of primitive models is based on the AST (Abstract Syntax Tree) used in LibClang [28]. The key concepts of the primitive model are in the following.

- **Label**: name of the primitive model.
- **Pattern**: the code patterns in AST (e.g., `DeclRefExpr`, `BinaryOperator '*'` and `CXXMemberCallExpr`) to define a behavior.
- **Object**: expected Mach port objects (e.g., `ipc_port_t` and `task_t`) to be processed in the primitive.
- **Entry_with_object**: Mach system call entries (e.g., `mach_port_set_context()`) for user space programs to invoke and Mach port objects (e.g., `task_t`) to be processed in the primitive. This is the output of the POP primitive searcher.

In XNU, different kinds of Mach port objects are stored in separate zones. Thus the attacker is only able to corrupt specific Mach port objects. Therefore, based on the type of corrupted Mach port objects, POP primitive searcher takes

Label, Pattern, and Object as inputs, then examines all possible Mach system call entries (i.e., Entry) to find possible POP primitives.

As an example, we assume an attacker has a send right to a dangling port and wants to get a RP (i.e., kernel memory read primitive) for the future exploitation. He or she can use the POP primitive searcher with the RP model (Listing 1) to find potential primitives. Specifically, Label denotes the type of this primitive. Object shows the expected Mach port objects (e.g., ipc_port_t and task_t) used in the primitive. Pattern is used to define the behavior of this type of primitive that transmits data from the kernel space to the user space. The POP primitive searcher outputs possible Mach system call entries (Entry) with suitable Mach port objects.

Listing 1: Kernel memory read primitive model

```
[Input]
Label: 'RP'
Object: ipc_port_t, task_t, proc_t, thread_t
Pattern: CallExpr convert_port_to_\S*\(.*\) |
         CallExpr port_name_to_\S*\(.*\) |
         CallExpr iokit_lookup_object_port\(.*\),
         BinaryOperator.*'='\n.*memberExpr |
         BinaryOperator.*'='\n^((?!BinaryOperator)[\s\S])
         {0,1000}ImplicitCastExpr.*\n.*MemberExpr|
         CallExpr.*\n^((?!CallExpr)[\s\S]){0,1000}
         DeclRefExpr.*copyout|
         CallExpr.*\n^((?!CallExpr)[\s\S]){0,1000}
         DeclRefExpr.*bcopy

[output]
         Entry_with_object:
         mach_port_get_context: ipc_port_t
         pid_for_task: task_t
         mach_port_get_attributes: ipc_port_t
         ...
```

B. Implementation

In the following, we provide an overview of our implementation.

Collecting resources POP primitive searcher relies on the XNU source code, the XNU kernel cache, and primitive models. The source code and the kernel cache are used to generate graphs for each Mach system call. Besides, primitive models are used to search POP primitives among these graphs. The XNU source code can be downloaded from the Apple's open source website [48]. The XNU kernel cache is a repository of pre-linked kernels. It contains exported function addresses and offsets of fields of kernel objects. Apple also releases all versions of the XNU kernel cache on its developer website [6].

Getting all Mach system call entries and Mach port objects For all possible system call entries using Mach messages (e.g., MIG system calls and Mach traps), the POP primitive searcher analyzes the XNU source code (e.g., MIG subsystem definition files and mach_trap_table) to generate a list of POP entries and related Mach port objects (see Table I) based on the Mach subsystems.

Generating AST for source code files POP primitive searcher modifies the Makefile of the XNU source code to replace the original compiling mode to the AST-dump mode (clang++ -Xclang -ast-dump -fsyntax-only) to generate AST for each source code file.

Generating Mach port-oriented control flow graph for each entry For each entry, POP primitive searcher constructs a Mach port-oriented control flow graph (**POC** in the paper) based on AST files. Note that the POC is a structured representation. All information about a function is bundled up in a class hierarchy.

The nodes in the POC are statements (Stmt), expressions (Expr), and declarations (Decl). The searcher recursively generates POCs for all functions from a system call entry. To solve the search complexity issue, POP primitive searcher only chooses code paths containing Mach port related objects to prune the graph. Specifically, it stores the structures of all *38* Mach port objects (e.g., task_t and thread_t) and links Mach system calls with these objects. Moreover, POP primitive searcher records possible code paths based on the object's member filed values. That means, through the POC graph, we can figure out kernel functions that will be executed by a Mach system call, as well as Mach port objects that are affected during the execution.

Searching matched primitives For each path in the POC, POP primitive searcher classifies the code path based on primitive models (Section IV-A). If the path has expected Mach port Objects and matches the code Pattern, POP primitive searcher will mark it as a potential primitive.

For instance, Figure 11 in the appendix shows a POC for the pid_for_task() system call. According to the DeclStmt of POC, the searcher firstly records three kernel objects, i.e., mach_port_name_t, proc_t, and task_t. Then, it records a port_name_to_task() CallExpr in the system call which transfers a mach_port_name_t object into a task_t Mach port object. Subsequently, the POP primitive searcher records code paths which contain the member filed values of the task_t object and other objects interacted with the task_t object. In particular, the code path invokes copyout(&pid, pid_addr, sizeof(int)) to copy kernel space data back to the user space address, which complies with our kernel memory read primitive model (Section IV-A).

C. Results

In our experiment, our searcher detected 713 potential POP primitives in XNU-6153.11.26. In the following, we will describe four different primitive models.

QP QP deals with randomization-based defenses (e.g., zone elements randomization and KASLR). The code pattern of QP model includes decision making statements (e.g., IfStmt and SwitchStmt) and return value assignment (e.g., ReturnStmt). Listing 2 shows the model of the querying primitive.

POP primitive searcher detected 436 querying primitives in nearly every Mach subsystem. For instance, in the TIME subsystem, by changing the ip_kobject field of the ipc_port_t object and invoking the clock_sleep_trap() function, the attacker can figure out the global address of the system clock and break KASLR (Section III-D). Moreover, some QPs can be used to bypass zone elements randomization. As we mentioned in Section II-B, with zone elements randomization, it is difficult for an attacker

Mach subsystem	# POP entries	Object type	Related object structure
RAW_PORT	39	IKOT_NONE	ipc_port_t
HOST	56	IKOT_HOST, IKOT_HOST_PRIV, IKOT_HOST_NOTIFY, IKOT_HOST_SEC	host_t, host_priv_t, host_security_t
PROCESSOR	18	IKOT_PROCESSOR, IKOT_PSET, IKOT_PSET_NAME	processor_t, processor_set_t, processor_set_name_t
TASK	105	IKOT_TASK, IKOT_TASK_NAME, IKOT_TASK_RESUME, IKOT_NAMED_ENTRY	task_t, task_name_t, task_inspect_t, ipc_space_inspect_t ipc_space_t, task_suspension_token_t, vm_map_t
MEMORY	30	IKOT_UPL, IKOT_MEM_OBJ, IKOT_MEM_OBJ_CONTROL	upl_t, memory_object_t, memory_object_control_t
THREAD	23	IKOT_THREAD	thread_t, thread_act_t, thread_inspect_t
DEVICE	88	IKOT_MASTER_DEVICE, IKOT_IOKIT_SPARE, IKOT_IOKIT_CONNECT, IKOT_UEXT_OBJECT, IKOT_IOKIT_IDENT	io_connect_t, io_object_t
SYNC	24	IKOT_SEMAPHORE, IKOT_LOCK_SET	semaphore_t, lock_set_t
MACH_VOUCHER	7	IKOT_VOUCHER, IKOT_VOUCHER_ATTR_CONTROL	ipc_voucher_t, ipc_voucher_attr_control_t
TIME	6	IKOT_TIMER, IKOT_CLOCK, IKOT_CLOCK_CTRL	mk_timer_t, clock_serv_t, clock_ctrl_t
MISC	49	IKOT_PAGING_REQUEST, IKOT_MIG, IKOT_XMM_PAGER, IKOT_XMM_KERNEL, IKOT_XMM_REPLY, IKOT_UND_REPLY, IKOT_LEDGER, IKOT_SUBSYSTEM, IKOT_IO_DONE_QUEUE, IKOT_AU_SESSIONPORT, IKOT_FILEPORT, IKOT_LABELH IKOT_WORK_INTERVAL, IKOT_UX_HANDLER, IKOT_ARCADE_REG	ledger_t, work_interval_t, mig_object_t, UNDReplyRef, auditinfo_addr, fileglob, arcade_register_t
MACH_TRAP	56	port_name_to_*()	thread_t, task_t, task_inspect_t, ipc_voucher_t, semaphore_t, host_t, work_interval_t
Sum	501	43	38

TABLE I: Classification of POP entries and Mach port objects (based on the MIG and Mach trap subsystems) in XNU-6153.11.26. Note that some object types may have multiple structures.

Listing 2: Querying primitive model

```
Label: 'QP'
Object: ipc_port_t, .*\_t
Pattern: CallExpr convert_port_to_\S*\(.*\)|
         CallExpr port_name_to_\S*\(.*\)|
         CallExpr iokit_lookup_object_port\(.*\),
         IfStmt([\d\D]{0,1000})DeclRefExpr.*_t|
         SwitchStmt([\d\D]{0,1000})DeclRefExpr.*_t,
         ReturnStmt([\d\D]{0,500})IntegerLiteral
```

to manipulate zone elements in a predictable way. Consequently, exploiting a buffer overflow vulnerability may destroy unexpected objects in the adjacent place, and make the system crash.

To bypass the zone elements randomization, an attacker can use a `QP` in the `RAW_PORT` subsystem with the `mach_port_kobject()` system call to check the type of an `ipc_port` object to know whether it is the one he/she just corrupted. Specifically, the `mach_port_kobject()` system call returns the value on the `io_bits` offset of the target object even if it is not a valid `ipc_port` object. Thus, the attacker can send lots of ports with the `IKOT_NONE` type through `MACH_MSG_OOL_PORTS_DESCRIPTOR` Mach message to the kernel and then trigger the memory corruption vulnerability to corrupt the type of an `ipc_port` to a particular type, e.g, `IKOT_TASK`. The exact type does not matter here, as long as it is different from the original one (`IKOT_NONE`). Then the attacker can use the `mach_port_kobject()` system call to check the type of a received port to see whether it is the one with the expected type (`IKOT_TASK`). If so, the corrupted kernel object has been found and will be further used to implement other primitives.

RP/WP `RP/WP` helps the attacker gain kernel memory read/write capability to limited or arbitrary kernel addresses. Listing 1 and Listing 3 describe the models of `RP` and `WP`, respectively. In our experiment, 29 `RP/WP` are detected by our searcher. Most of them are found in the `TASK` and `MACH_TRAP` subsystems. For example, `mach_port_set/get_*()` can gain limited kernel memory read/write through a dangling port [1]. Specifically,

[1]A port referring to a freed `ipc_port` object is called a *dangling* port. A `ipc_port` object contains a set of member fields pointing to the kernel memory.

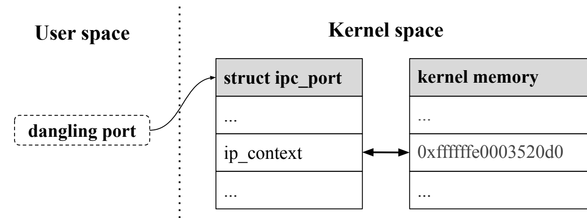

Fig. 6: Limited kernel memory R/W through mach_port_set/get_*() [13].

the `ip_context` field in the `ipc_port` object is used to associate a user space pointer with a port (see Figure 1). By applying `mach_port_get_context()` and `mach_port_set_context()` to a dangling port, the attacker can read and write a 64-bit value from and to a kernel address, which points to the freed `ipc_port` object plus the `ip_context` offset (Figure 6).

Listing 3: Kernel memory write primitive model

```
Label: 'WP'
Object: ipc_port_t, task_t, proc_t, thread_t
Pattern: CallExpr convert_port_to_\S*\(.*\)|
         CallExpr port_name_to_\S*\(.*\)|
         CallExpr iokit_lookup_object_port\(.*\),
         BinaryOperator.*'='\n.*memberExpr|
         BinaryOperator.*'='\n^((?!BinaryOperator)
         [\s\S]){0,1000}.*MemberExpr|
         CallExpr.*\n^((?!CallExpr)[\s\S])
         {0,1000}DeclRefExpr.*copyin|
         CallExpr.*\n^((?!CallExpr)[\s\S])
         {0,1000}DeclRefExpr.*bcopy
```

PPP If the attacker can obtain a send right to privileged ports, he/she can enlarge the attack surface or even control the whole system through `PPP`. The key component of the `PPP` model is the `Object` type. That's because `PPP`s process privileged `ipc_port` objects, including `host_priv_t` and `kernel_task`. The model of the privileged purpose primitive is defined in Listing 4.

In total, our searcher found 176 privileged purpose primitives existing in `HOST`, `PROCESSOR` and `TASK` subsystems. For instance, with a send right to the `host_priv` port, an attacker can gain send rights to other powerful ports (e.g., `processor_set` port). Moreover, with a send right

Listing 4: Privileged purpose primitive model

```
Label: 'PPP'
Object: host_priv_t, kernel_task, vm_map_t
Pattern: CallExpr convert_port_to_\S*\(.*\)|
         CallExpr port_name_to_\S*\(.*\)|
         CallExpr iokit_lookup_object_port\(.*\)
```

to the `kernel_task` port, the attacker can leverage `PPPs` (i.e., `mach_vm_*()` system calls) to manipulate the kernel memory.

Other attack primitives Beside the previous three primitives, there exist other primitives that can be leveraged to launch further attacks like jailbreaking the device. Note that these primitives are not necessary for a POP attack. In this section, we use the arbitrary code execution primitive (`ACEP`) as an example. The key component of the `ACEP` primitive model is function pointer related code patterns (e.g., `CXXMemberCallExpr`). The model of the `ACEP` is defined in Listing 5.

Listing 5: Arbitrary code execution primitive model

```
Label: 'ACEP'
Object: io_object_t, io_connect_t, clock_t, .*
Pattern: CallExpr convert_port_to_\S*\(.*\)|
         CallExpr port_name_to_\S*\(.*\)|
         CallExpr iokit_lookup_object_port\(.*\),
         CallExpr([\d\D]{0,500})\
         ImplicitCastExpr([\d\D]{0,500})->|
         CXXMemberCallExpr([\d\D]{0,500})->
```

Our searcher detected 72 arbitrary code execution primitives. Most of them are found in the `DEVICE` subsystem (e.g., the `iokit_user_client_trap()` discussed in Section III-D). The `TIME` subsystem also contains some `ACEPs`. For instance, the `clock_get_time()` system call first gets the `cl_ops` structure pointer and then invokes time related functions inside the `cl_ops` structure to get the time information (Listing 9 in Appendix). Moreover, the `cl_ops` structure pointer is initialized to point to the `clock_list[]` array (a global variable) stored in a fixed place in the kernel. Consequently, the attacker can modify the data of the `clock_list[]` array or the clock object to achieve arbitrary kernel code execution.

V. PUSH

To mitigate the POP attack, we propose a framework called *Port Ultra-SHield* (PUSH) on macOS. In this section, we will present the system design and implementation of the PUSH framework.

A. System Design

PUSH first searches different types of potential POP primitives previously discussed in Section IV, and then enforces security policies to maintain the integrity of kernel objects that can be corrupted by attackers. Figure 7 shows the overall system design. In particular, PUSH consists of the following main components:

- **POP primitive searcher** (Section IV) It is a tool to automatically locate potential POP primitives and output Mach system calls with related Mach port objects based on primitive models. Specifically, our system generates Mach port-oriented control flow graphs (POC) and finds different primitives.
- **PUSH policy generator** It is a framework to generate PUSH policy rules to enforce the integrity of Mach port objects before and/or after particular Mach system calls. For each Mach port object, PUSH policy generator can take different kinds of actions (e.g., checking object address and object data).
- **PUSH checker** It is a kernel extension to deploy PUSH examiners. To make the system more flexible, security rules can be updated at runtime.

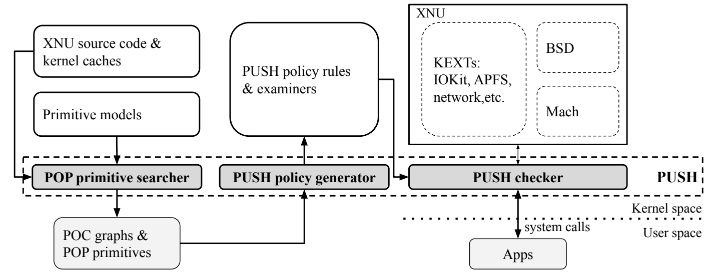

Fig. 7: The overall design of PUSH.

We will illustrate PUSH policy generator and PUSH checker in the following sections.

B. PUSH Policy Generator

Based on the result of POP primitive searcher, PUSH policy generator knows the Mach port objects that need to be protected, as well as system call names. Moreover, it selects the appropriate examiner to protect these objects. However, examiners (currently, four examiners were implemented) themselves need to be developed manually.

To manage generated rules, we implemented a PUSH policy framework. The basic structure of a PUSH policy is defined in the Listing 6. The `push_object_name` and `push_entry_name` specify the name of protected kernel object and related system call name. The `mpc_ops` is a pointer to the registered callback functions in the `PUSH_policy_ops` structure. The callback functions are examiners for different types of events in the life cycle of a kernel object, such as creation, initialization, transformation, and destruction.

Listing 6: PUSH policy definition

```
struct push_policy_conf {
    const char *push_entry_name;
    const char *push_object_name;
    // operation vector
    const struct PUSH_policy_ops *mpc_ops;
};

struct PUSH_policy_ops {
    object_address_examiner_t *koa_examiner;
    object_querying_examiner_t *koq_examiner;
    kernel_task_examiner_t *ktv_examiner;
    object_data_examiner_t *kot_examiner;
    // extensible policy rules
    ...
};
```

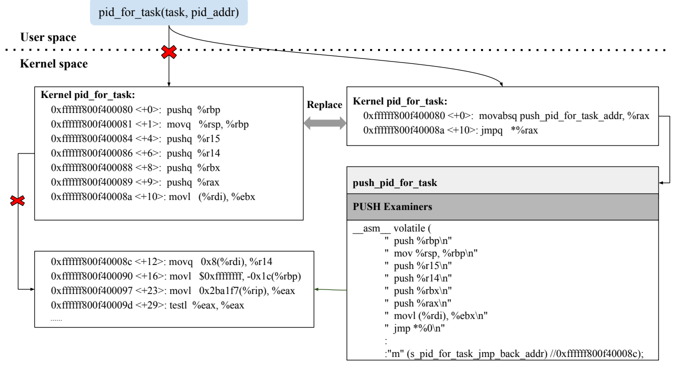

Fig. 8: `pid_for_task()` instrumentation and checking [13].

To mitigate the POP attack, our current prototype supports four different kinds of examiners illustrated in the following.

1) Kernel object address examiner: For old iOS/macOS devices, the CPU does not support PAN (Privileged Access-Never)/SMAP [39], which enforces the rules that the kernel code cannot access user space memory. However, without PAN/SMAP, even on the latest iOS and macOS system, the attacker can still directly make a dereference for a `mach_port_name_t` object in the kernel space to a fake `ipc_port` object in the user space. This significantly reduces the difficulty of the attack, because the attacker does not need to craft a counterfeit object in the kernel space and guess its address.

We implemented the kernel object address examiner for the `ipc_object` when the hardware-based SMAP is not available. The examiner checks dereference operations for `mach_port_name_t` objects in related system calls. If the address of the dereferenced kernel object is not in the kernel zone area (i.e., `addr >= zone_map_min_address && addr + obj_size <= zone_map_max_address`), the examiner will return immediately with a warning or an error, according to the user's configuration.

2) Kernel object querying examiner: In Section IV-C, we discussed an accessory that uses the POP querying primitives to break freelist randomization and KASLR. This attack relies on querying the return values of system calls or the value of the field of specified kernel objects. To mitigate such a brute-force attack, the kernel object querying examiner counts the frequency of the `ERROR` return value or specified kernel object (e.g., same port name with different object types) that are accessed in a time period, and then triggers different operations based on the user's configuration (e.g., a warning or an error return value).

3) Kernel task examiner: In iOS 11 and macOS 10.12, Apple deployed a Mach port related mitigation called `task_conversion_eval()` [49] to protect the kernel's task port. However, for `mach_vm_*()` related system calls discussed in Section III-D, the most significant unit is the task's `map` structure of an `ipc_port` object. An attacker could craft a counterfeit `ipc_port` object with the kernel task's `map` structure to control the kernel memory and bypass `task_conversion_eval()` mitigation [21]. To this end, the kernel task examiner instruments related system calls and compares the processing `map` structure with the one of the kernel tasks. If the caller process does not belong to the kernel (`current_task()` != `kernel_task`) and the target `ipc_port` object has the same `map` structure with the kernel task, the examiner will trigger configured operations.

4) Kernel object data examiner: For the purpose of reducing our system's performance overhead, there is no integrity check for the `ipc_port` objects and its sub-objects. For instance, `pid_for_task()` invokes `get_bsdtask_info()` and `proc_pid()` to get the process info and the `pid` number of the target process. Nevertheless, the XNU kernel does not check whether the object is a real process object or not, and just returns the value in a fixed offset. Due to this reason, the type-confusion attack can be launched to achieve an arbitrary kernel memory read primitive through `pid_for_task()` (Section III-D).

To mitigate such an attack, the kernel object data examiner checks the member fields of the object when it is being invoked and enforces its integrity. In our system, there are three types of object data integrity check. They include the checks for `ip_kobject`, `ip_receiver`, and `ip_requests` in the PUSH examiner. For instance, the `ip_requests` field of the `ipc_port` object is used in the `mach_port_get_attributes()` system call. By providing a flavor value of `MACH_PORT_DNREQUESTS_SIZE`, the system call will return the value of `fakeport->ip_requests->ipr_size->its_size`. Thus the attacker can use this system call to retrieve kernel memory information by counterfeiting a port with a calculated `ip_requests` structure. PUSH will check the integrity of the `ip_requests` object to ensure it is valid.

Extensible policy rules and examiners. With the development of the XNU kernel, new types of kernel objects and system calls will emerge. Accordingly, there is a high probability that new attack primitives will be located and exploited by attackers. To make our system defend against new attack primitives, it can scan the new XNU source code and generate new rules. After that, the new rules could be loaded by PUSH kernel extension, without changing the source code of the extension itself.

C. PUSH Checker

Our system needs to find reliable code points to execute the examiners. Unfortunately, the KAuth [7] and the MAC system in the XNU kernel cannot be used by our system. Specifically, the KAuth [7] kernel subsystem exports a kernel programming interface (KPI) that allows third-party kernel developers to authorize actions within the kernel. However, the operation set is limited. Most POP primitives could not be covered by supported operations. On the other hand, the MAC framework is more powerful than KAuth. Its callouts exist in all system calls and every single operation can be intercepted. But the MAC framework is private and can only be used by Apple. Moreover, the rules are hardcoded in the code of the XNU kernel, and new rules cannot be updated without a kernel replacement.

To solve these issues, we propose PUSH checker, a kernel extension which dynamically deploys PUSH policy rules using the code instrumentation technique. PUSH checker can be

loaded into the kernel through kextutil [26] which does not require a kernel code modification. Based on PUSH rules, PUSH checker replaces the original code entry of the target system call into a trampoline (i.e., a jump instruction to another code path). The trampoline jumps to the examiner stored in PUSH checker. Then, the examiner verifies the integrity of the target kernel object based on PUSH rules. If the check passes, it jumps back to the original function. Figure 8 in Appendix shows a real example of the instrumentation for the pid_for_task() system call and the detour routine to the corresponding checkers.

Note that PUSH only adds limited verification code before invoking a system call. The verification code does not contain dangerous operations like kernel heap memory allocation and modification. Thus, it will not lead to more POP primitives since it does not add new POP entries. In addition, the self-protection is needed to protect the loaded rules and critical PUSH data in the kernel memory. This is achieved with a read-only memory protection through the page table. First, a read-only memory is enough to protect the critical PUSH data since attackers only have limited memory corruption capability. They cannot directly manipulate read-only memory regions. Second, by using our PUSH system, the POP attack can be defeated. Thus attackers cannot achieve the arbitrary code execution capability to change the memory protection property of the page table.

VI. Evaluation

In this section, we evaluate the effectiveness and the performance overhead of our prototype.

A. *Effectiveness*

To evaluate the effectiveness of our system, we first select representative vulnerabilities and exploits in the wild. In particular, we first performed a statistic study of the vulnerabilities existed in XNU. To this end, we implemented a crawler to dump the information published on the Apple security updates [8], and then selected kernel vulnerabilities, ranging from macOS Sierra 10.12 to macOS Catalina 10.15. Since our goal is to evaluate the effectiveness of PUSH framework in protecting the macOS system in the case of vulnerabilities, we further collect *all* working exploits from public bug report platforms, e.g., Google's project-zero bug reports [37], and ExploitDB [15]. Finally we collected 11 kernel vulnerabilities and 18 public exploits [2] to evaluate the effectiveness of our system. Note that some exploits are developed for iOS system. We manually ported them into the macOS platform.

In the experiment, we first ensure that each exploit works on corresponding systems. Then we deploy the PUSH framework and run the exploit again to check whether our system detects and blocks the attack. The experiment result shows PUSH provides deterministic protection for every vulnerability and blocks each attempt to exploit the system (see Table II).

In the following, we will use three cases to explain how our system blocks the exploit and protects the XNU kernel.

Case study 1: Yalu exploit We have discussed the Yalu exploit in Section III-D. It counterfeits the kobject field of ipc_port object with IKOT_CLOCK type and calls the clock_sleep_trap() system call to locate the address of the global system clock with a brute force attack. By using the kernel object querying examiner we proposed in Section V-B2, PUSH will detect the brute force process after a few failed attempts. Moreover, because the fake ipc_port object is allocated in the user space, the kernel object address examiner (Section V-B1) will block the attempt of the exploitation in the first place.

Case study 2: Voucher_swap exploit CVE-2019-6225 [23] is a MIG reference counting vulnerability, which affects iOS 12 and macOS 10.14. The code exists in function task_swap_mach_voucher(). The automatically generated MIG code is in _Xtask_swap_mach_voucher() function (Listing 11 in Appendix). The input value of in_out_old_voucher is a voucher reference owned by task_swap_mach_voucher(). Unconditionally overwriting it without calling ipc_voucher_release() will leak a voucher reference. In addition, the value new_voucher is not owned by task_swap_mach_voucher() and it is being returned in the output value of in_out_old_voucher. It will consume a voucher reference that task_swap_mach_voucher() does not own. Thus, an attacker can leak a reference on a voucher by calling task_swap_mach_voucher() with the voucher as the third argument, and drop a reference on the voucher by passing the voucher as the second argument.

Accordingly, the exploit makes a dangling voucher through task_swap_mach_voucher(). Moreover, by crafting an OOL (out-of-line) port message and refilling it to the freed voucher zone, the exploit could modify the overlapping voucher's iv_refs field, which changes one port pointer in the OOL port message and makes it point to a fake ipc_port object. After that, the exploit gets a send right to the fake ipc_port object through receiving OOL port message. Then, it builds a fake task object to gain an arbitrary kernel memory read primitive via pid_for_task(). Nevertheless, according to our kernel object data examiner (Section V-B4), PUSH checks the member fields of the object when it is being invoked. The tricks used in pid_for_task() system call should only return a valid pid number of the current process which makes the exploit chain failed to get significant kernel information.

Case study 3: Oob_timestamp exploit CVE-2020-3837 [18] is a vulnerability affecting iOS 13 and macOS 10.15. The vulnerable is in the AGXCommandQueue::processSegmentKernelCommand() method of the IOAcceleratorFamily kernel driver. The checking of the size in this method to parse the IOAccelKernelCommand parameter from the user space is incorrect. It allows the attacker to write a 8-byte timestamp data past the end of a shared memory buffer.

Consequently, the oob_timestamp exploit uses the out-of-bounds timestamp data to corrupt the size of an ipc_kmsg object of a Mach message. When the message is destroyed, it will free a subsequent out-of-line ports array, allowing the attacker to reallocate the array with controlled data and receive a Mach port of a fake ipc_port object in the user space. Moreover, the fake ipc_port object points

[2] All of our collected exploits can be downloaded at [1].

TABLE II: Detailed information of representative vulnerabilities and exploits used in our evaluation

macOS/iOS version	Vulnerability (CVE number)	Public exploit	Targeted Mach port object
macOS 10.12/iOS 10	CVE-2016-4669	Phoenix [35]	IKOT_NONE, IKOT_IOKIT_CONNECT, IKOT_TASK
	CVE-2016-7644	Mach_portal [47]	IKOT_NONE, IKOT_TASK
	CVE-2017-2370	Yalu [51], Mach_voucher_macOS [32]	IKOT_CLOCK, IKOT_TASK, IKOT_IOKIT_CONNECT
macOS 10.13/iOS 11	CVE-2017-13861	Async_wake [21]	IKOT_NONE, IKOT_TASK ,IKOT_HOST_PRIV
	CVE-2018-4150	Bfp-filter-poc [16]	IKOT_NONE, IKOT_CLOCK, IKOT_TASK, IKOT_IOKIT_CONNECT
	CVE-2018-4241	Multi_path [46], Multipath_kfree [34]	IKOT_NONE, IKOT_TASK
	CVE-2018-4243	Empty_list [33]	IKOT_NONE, IKOT_TASK
	CVE-2018-4344	Spice [38], Treadm1ll [42]	IKOT_NONE, IKOT_TASK, IKOT_IOKIT_CONNECT
macOS 10.14/iOS 12	CVE-2019-6225	Voucher_swap [23], v3ntex [45] Machswap [30], CVE-2019-6225-macOS [11]	IKOT_NONE, IKOT_VOUCHER , IKOT_TASK
	CVE-2019-8605	SockPuppet [24]	IKOT_NONE, IKOT_HOST, IKOT_TASK
macOS 10.15/iOS 13	CVE-2020-3837	Oob_timestamp [18], Time_waste [41]	IKOT_NONE, IKOT_TASK , IKOT_HOST_PRIV

to a fake `task` which has the same `map` field as that of the kernel's `task`. Thus, the attacker could use `mach_vm_*()` system calls to manipulate the kernel memory. Our kernel task examiner of the PUSH framework (Section V-B3) will check the target `map` structure of the caller process with the kernel, and then stop the user space programs from manipulating the kernel memory.

Note that we proposed and implemented PUSH before the releasing of `voucher_swap` and `oob_timestamp` exploit. This demonstrates the effectiveness of our system to block new exploits.

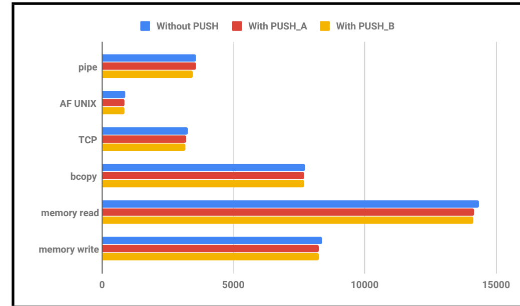

Fig. 9: Communication bandwidths in MB/s (bigger is better).

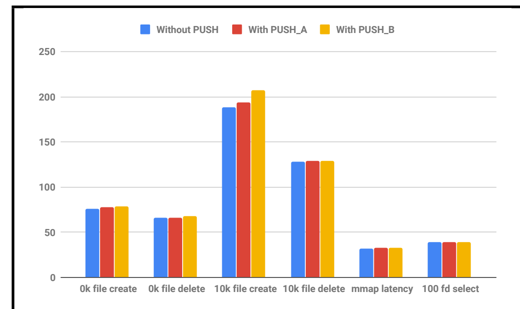

Fig. 10: File & VM system latencies (times in microseconds, smaller is better).

B. Performance

Benchmarks and evaluation setup To evaluate the overhead induced by PUSH, we run several benchmark programs: LMBench [29], wrk HTTP benchmarks [44] for Apache, and Sysbench benchmarks [40] for MySQL. We first run programs on the macOS system without PUSH. Then, we deploy PUSH_A (without kernel object address examiner) and PUSH_B (with kernel object address examiner) on the system [3]. After that, we run benchmark programs again. We run each benchmark 10 times and calculate the average execution time. Our experiments are carried on a MacBook Air with 1.3 GHz Intel Core i5 CPU and 4 GB memory, with a few different XNU versions crawled from the Apple developer website[6], ranging from macOS Sierra (10.12) to macOS Catalina (10.15).

Instrumentation points The performance of PUSH is mainly correlating with the number of instrumentation points (IPs). Specifically, the performance will become worse when more IPs are deployed in the system. The number and the location of IPs for PUSH are shown in the following:

- QP: PUSH added 4 IPs to prevent attackers from guessing global kernel object addresses.
- RP/WP: PUSH added 8 IPs to perform the data integrity check for the `ipc_port` object and related `kobjects`.
- PPP: PUSH added 2 IPs for `mach_vm_*()` and `host_*()` related system calls to check the `host_priv` and `kernel task` Mach port objects.
- ACEP: currently, PUSH does not add IPs for ACEP. We leverage existing CFI (e.g., LLVM-CFI [27] and PAC) to defend the ROP gadgets to gain this capability. Noting that CFI cannot be leveraged to defend POP.
- Kernel object address examiner (on PUSH_B): for devices without the SMAP protection, PUSH added 43 IPs for address verification in port name dereference functions (e.g., `port_name_to_*()` and `convert_port_to_*()`).

Note that 13 IPs are used to protect POP primitives in real world exploits. Other IPs can also make POP exploits ineffective. However, this *over-protection* will not cause problem to our system, except that it will cause more performance overhead.

LMBench Table III shows the result of processor/processes activities of LMBench benchmark programs. `PUSH_A`'s over-

[3] For some modern MacBooks with a hardware based SMAP protection or systems with `zone_require()` protection, the kernel object address examiner is not necessary.

TABLE III: Processor/processes activities (times in microseconds, smaller is better).

OS	null I/O	open close	fork proc	exec proc	sh proc
Without PUSH	1.43	16.8	682	3778	7310
With PUSH_A	1.43 (0%)	17.0 (1.2%)	686 (0.5%)	3807 (0.8%)	7345 (0.5%)
With PUSH_B	1.44 (0.6%)	17.1 (1.7%)	693 (1.6%)	3851 (1.9%)	7393 (1.1%)

TABLE IV: Wrk benchmark experiment results for the Apache HTTP server system. In the case of the same number of threads (12) and running time (120s), the more HTTP requests the system sends, the better the system performs. PUSH introduces 1.32% overhead for HTTP request generation and 1.28% overhead for the data transfer.

Test Model	Time	Without PUSH	With PUSH_B	Overhead
Request generation	120s	7148.06 reqs/sec	7054.93 reqs/sec	1.32%
Data transfer	120s	2.37 MB/s	2.34 MB/s	1.28%

head is from 0% to 1.2% and `PUSH_B`'s overhead is from 0.6% to 1.9%. The result is expected. The benchmark programs that hit more IPs will have a larger overhead. Moreover, the results of communication bandwidths (showed in Figure 9) and file & VM system latencies (showed in Figure 10) are very similar to the result of processor/processes activities. In general, the system only has an average of 1% overhead (without kernel object address examiner) and 2% overhead (with kernel object address examiner) for LMBench benchmark programs.

Apache and MySQL Apache is a free and open-source cross-platform web server. The `wrk` is a modern HTTP benchmark tool capable of generating HTTP client requests. In the experiment, we use `wrk` to stress test the Apache http server running on macOS with PUSH. Table IV shows the result. In the case of the same threads and running time, the more HTTP requests the system sends, the better the performance of the system. PUSH presents 1.32% overhead for HTTP request generation and 1.28% overhead for data transfer.

MySQL is an open-source relational database management system (RDBMS). Sysbench is a multi-threaded benchmark tool based on LuaJIT. It is frequently used for database benchmarks. In the experiment, we use Sysbench to generate test cases of on-line transaction processing for MySQL. Table V shows the result. In the same running time (1,800s), the more the system operates, the better the system performs. PUSH has 0.96% overhead for the READ_ONLY test model with 155 instrumentation point hits and 1.87% overhead for the READ_WRITE test model with 548 instrumentation point hits.

Besides, this experiment is tested on the system with full PUSH policies loaded. Users could configure lightweight rules for specific kernel objects to further reduce the performance overhead.

VII. Real deployment

The PUSH system has been deployed in Alibaba Group with more than *40,000* macOS devices for nine months (since 2019.8). It successfully detected two mutated exploits (based on voucher_swap [23] and oob_timestamp [18]) and one new exploit. The new exploit is used in an E-mail phishing attack. Specifically, the attacker sent a phishing email with a malicious app as the attachment. The app is disguised as a PDF document that will execute an AppleScript [19] once opened. The script downloads a real PDF file and opens it through the `Priview` app. Then it downloads an attack payload. The attack payload uses a zero-day memory corruption vulnerability in the Graphics kernel driver to escalate its privilege and leaves a persistent backdoor with root privilege in the system (by leaving a trojan file written by `Go` language in the `/usr/local/bin/` folder and a `plist` launch file in the `/Library/LaunchDaemons/` folder). Because the attack payload uses POP primitives (e.g., pid_for_task() and mach_vm_write()) to gain kernel memory read/write capability and change its permission, it was detected by the PUSH system. We have reported this zero-day vulnerability to Apple [4].

Note that for each alert reported by the PUSH framework, the detailed execution context information including the process information and the executable program will be transferred to a deployed EDR (Endpoint Detection and Response) client. The EDR client will upload this information to a remote server for further manual verification. Fortunately, we have not found any false positives of the reported alerts yet.

However, iOS and watchOS cannot load third-party kernel extensions. To deploy our system on these devices, we need to first jailbreak the devices. For instance, we can leverage iBoot exploits (e.g., checkra1n [10]) to jailbreak old Apple devices (from iPhone 5s to iPhone X). In fact, it is much easier for Apple to re-implement a similar PUSH system for iOS. For instance, the rules used in our system can be transparently translated to corresponding MAC rules of the XNU kernel by Apple.

We are happy to know that after reporting our discovery and discussing the PUSH framework with their security team (follow-up id: 707542859), Apple will *deploy security enhancements against the Mach port issue in a future release of iOS and macOS*. In fact, recently released systems (iOS 13 and iOS 14 beta) have deployed new protective measures (e.g., `zone_require()` and data-PAC for kernel objects) that are similar with our kernel object address examiner (see Section V-B1) and kernel object data examiner (see Section V-B3).

VIII. Discussion

In this section, we discuss limitations and possible improvements of our system.

First, our work only focuses on the Mach port kernel object which is a high value target for attackers due to the capability they can achieve after corrupting it. However, there may exist other similar kernel objects that could be exploited. Second, due to the search complexity, the POP primitive searcher only focuses on code paths that contain Mach port related objects. It may miss some kernel objects and functions, leading to the incomplete call graph and data flow graph. Moreover, the POP primitive searcher detects POP primitives based on primitive models. In the current prototype, all primitive models are manually developed. They highly rely on the expert experience. Consequently, our searcher may lose some potential POP primitives. Thus, one possible improvement is to leverage the program analysis technique (e.g., CPU emulator framework [43]) to improve the completeness of the primitive searcher.

[4]This zero-day vulnerability is still under Apple's internal investigation. Apple requested us to keep it confidential. We will update the CVE number and release a POC of the vulnerability at this link [1] after it is fixed by Apple.

TABLE V: Sysbench benchmark experiment results for the MySQL database management system. `Read Ops`, `Write Ops`, `Other Ops` and `Total Ops` columns present the number of related SQL querying operations. `IP Hits` shows the number of PUSH instrumentation points that were hit. In the case of the same `Running Time` (1,800s), the bigger number of the system operation means the better performance. PUSH introduces 0.96% overhead for the READ_ONLY test model with 155 instrumentation point hits and 1.87% overhead for the READ_WRITE test model with 548 instrumentation point hits.

Test Model	Time	Without PUSH					With PUSH_B					Overhead
		Read Ops	Write Ops	Other Ops	Total	IP Hits	Read Ops	Write Ops	Other Ops	Total Ops	IP Hits	
READ_ONLY	1,800s	2,891,784	0	413,112	3,304,896	0	2,864,246	0	409,178	3,273,424	155	0.96%
READ_WRITE	1,800s	1,399,048	399,728	199,864	1,998,640	0	1,373,288	392,368	196,184	1,961,840	548	1.87%

Besides, PUSH cannot mitigate all kinds of POP primitives. For instance, querying primitives use error return values to obtain information. It is similar to the side-channel attack, which is based on information retrieved from the implementation of a computer system, rather than the weaknesses in the implemented algorithm itself (e.g. vulnerabilities). Kernel object querying examiner we proposed in Section V-B2 may not eliminate all of the querying primitives.

Moreover, PUSH does not intend to prevent the arbitrary code execution primitive, because it is not a necessary step to lunch the POP attack in the first place. For instance, critical kernel data could be changed (through `PPP`) to escalate privileges, instead of executing arbitrary code. A common POP attack usually follows the steps described in Section III-C. Although an attacker could omit some steps by chaining multiple vulnerabilities, the attack can still be detected by our system, unless the attacker does not use *any protected POP primitives*.

The evaluation on public exploits demonstrated the effectiveness of our system to block them. However, it cannot prove that our system can block every new attacks. Fortunately, PUSH supports extensible policy rules with examiners to prevent new threats in the first place. As long as the (new) exploit uses protected primitives, it can still be detected and blocked by our system. This has been demonstrated by the discovery of a zero-day vulnerability and the corresponding exploit (Section VII). In addition, we will keep updating the collection of public POP exploits [1] and practical POP attack primitives.

IX. RELATED WORK

PUSH is a generic kernel object protection framework for the XNU kernel, providing Mach port kernel object integrity to prevent the POP attack. In this section, we will illustrate other systems related to our work.

Szekeres et al. [67] systematically studied the memory corruption attack, from the potential reasons to multiple defense mechanisms. They also discussed why most protection mechanisms are not deployed in practice. They suggest that only solutions whose overhead is reasonable can be really deployed. Our proposed solution only incurs less that 2% overhead. This contributes the real deployment of our system.

Abadi et al. [53] proposed the concept of control-flow integrity. It enforces a security policy that the control flow of a program cannot be hijacked. In particular, it computes legitimate control flow transfers and then monitors the program execution so that the control flow transfer is in the predefined set. CFI in general is an excellent security mechanism. However, its effectiveness relies on the accuracy of the control flow graph. Unfortunately, to obtain a complete and sound CFG is sill a challenging task, due to the imperfect point-to analysis.

The Counterfeit object-oriented programming (COOP) [63] is an attack of CFI systems. COOP proofs lots of defenses (Code-Pointer Integrity [60], T-VIP [57], vfGuard [61], and VTint [70]) that specifically target C++ are vulnerable and Turing-complete attacks can be built with existed virtual functions in C++. The reason is that many of these defenses do not consider object-oriented C++ semantics. COOP creates counterfeit objects that use existed virtual tables chosen by attackers. Conceptually, the basic idea of our POP attack is similar to the COOP attack. However, they face different technical challenges. POP works for the XNU kernel. It needs to automatically search for different types of attack primitives through Mach port objects. This requires our system to understand the interactions between user space programs and the XNU kernel and take a Mach port aware way to locate code gadgets. COOP works for C++ programs. It leverages the virtual functions inside a user space program to construct the gadget, which does not need to consider the semantics of underlying kernel. Moreover, to ease the code-reuse attack, complex gadget building frameworks like Newton [68] and BOPC [59] are proposed. The proposed POP primitive searcher in this paper could also be leveraged to ease the POP attack.

Accordingly, Veen et al. proposed TypeArmor [69], a detection and confinement solution to mitigate advanced code-reuse attacks (e.g., COOP [63]). The system relies on binary-level static analysis to derive both target-oriented and callsite-oriented control-flow invariants. Then it applies security policies at runtime. However, as previously discussed, the inaccuracy of the call graph may leave loopholes for attackers.

Because CFI can only enforce control flow integrity, attackers are shifting to data-only attacks and proposed the data oriented programming (DOP) [58]. DOP constructs expressive non-control data exploits from arbitrary X86 programs. With a single memory error, non-control data attack can achieve Turing-complete computation using data-oriented programming. To defend against the data only attack, Chen [56] proposed a new compartmentalization mechanism to treat all memory inside the system as sensitive. Moreover, Schlesinger et al. [62], and Gaye et al. [55] proposed policies like data confidentiality and integrity (DCI) that selectively protects sensitive data. Song et al. [65] designed KENALI to mitigate privilege escalation attacks based on data-flow integrity. However, POP is a new kind of DOP attack that targets kernel objects. The proposed solutions may not work unless they could protect the integrity of `ipc_port` kernel objects. We have summarized the attack of POP and proposed a defense mechanism called XNU Kernel Object Protector (`XKOP`) to protect kernel objects in the XNU [13]. However, the talk did not discuss the way to automatically find POP gadgets, as well

as the implementation detail of XKOP. Our paper proposed the methodology to detect POP gadgets and reported the result of newly detected root exploits after deploying the system.

X. Conclusion

In this paper, we summary a new attack called *(Mach) Port-oriented Programming* (POP) that leverages multiple Mach ipc_port kernel objects to bypass existing mitigations. Consequently, a defense mechanism called *Port Ultra-SHield* (PUSH) is proposed to protect the integrity of Mach port objects in the XNU kernel with small performance overhead (2%). The PUSH system has been deployed in a leading company with more than $40,000$ devices. The evaluation of 18 public exploits and one zero-day exploit demonstrate the effectiveness of the proposed system. We have discussed this framework with Apple (follow-up id: 707542859). They appreciate our efforts to improve the security of their products and have deployed security enhancements against the Mach port issue in recently released XNU systems.

Acknowledgement

We sincerely appreciate the shepherding from Dr. Brendan Saltaformaggio and valuable comments from anonymous reviewers. We would also like to thank Tao Lei, Haitao Yu, Yanbin Dong, Xuesong Chen, Xin Luo, Yang Hua, Shuohui Lin, Ke Liu, Tuo Zhou, and Lei Qian from Alibaba Group. This work was supported in part by National Natural Science Foundation of China under Grant 61872438, 61772308, 61972224 and U1736209, the Fundamental Research Funds for the Central Universities (K20200019), and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] A collection of POP exploits. https://github.com/zhengmin1989/POP_AND_PUSH.
[2] Address space layout randomization, pax-team. https://pax.grsecurity.net/docs/aslr.txt.
[3] Advanced return-into-lib(c) exploits (pax case study) - phrack magazine. http://phrack.org/issues/58/4.html.
[4] Alibaba group. https://www.alibabagroup.com/en/global/home.
[5] american fuzzy lop. http://lcamtuf.coredump.cx/afl/.
[6] Apple Developer Website. https://developer.apple.com/.
[7] Apple Kernel Authorization (KAuth). https://developer.apple.com/library/archive/technotes/tn2127/_index.html.
[8] Apple Security Updates. https://support.apple.com/en-us/HT201222.
[9] Behind the scenes of ios and mac security, apple. https://i.blackhat.com/USA-19/Thursday/us-19-Krstic-Behind-The-Scenes-Of-IOS-And-Mas-Security.pdf.
[10] checkra1n, iPhone 5s iPhone X, iOS 12.3 and up. https://https://checkra.in/.
[11] CVE-2019-6225-macOS, TrungNguyen1909. https://github.com/TrungNguyen1909/CVE-2019-6225-macOS.
[12] Data Execution Prevention, Microsoft. https://docs.microsoft.com/en-us/windows/desktop/memory/data-execution-prevention.
[13] Eternal war in xnu kernel objects, blackhat europe 2018. https://i.blackhat.com/eu-18/Thu-Dec-6/eu-18-Zheng-Eternal-War-in-XNU-Kernel-Objects.pdf.
[14] Examining Pointer Authentication on the iPhone XS, Brandon Azad. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.
[15] ExploitDB. https://www.exploit-db.com.
[16] exploit for bpf filter, littlelailo. https://github.com/littlelailo/incomplete-exploit-for-CVE-2018-4150-bpf-filter-poc-.
[17] Heap Feng Shui in JavaScript, BlackHat Europe 2007. https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.
[18] iiOS/macOS: OOB timestamp write in IOAccelCommandQueue2::processSegmentKernelCommand(), Google PJ0. https://bugs.chromium.org/p/project-zero/issues/detail?id=1986.
[19] Introduction to applescript language guide. https://developer.apple.com/library/archive/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html.
[20] ios security guide. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.
[21] iOS/MacOS kernel double free due to IOSurfaceRootUserClient not respecting MIG ownership rules. https://bugs.chromium.org/p/project-zero/issues/detail?id=1417.
[22] iOS/MacOS kernel double free due to IOSurfaceRootUserClient not respecting MIG ownership rules, Google. https://bugs.chromium.org/p/project-zero/issues/detail?id=1417.
[23] iOS/macOS: task_swap_mach_voucher() does not respect MIG semantics leading to use-after-free, Google PJ0. https://bugs.chromium.org/p/project-zero/issues/detail?id=1731.
[24] iOS/macOS: XNU: Use-after-free due to stale pointer left by in6_pcbdetach, Google PJ0. https://bugs.chromium.org/p/project-zero/issues/detail?id=1806.
[25] Kernel Integrity Protection, Apple. https://support.apple.com/guide/security/kernel-integrity-protection-secb1caeb4bc/1/web/1.
[26] kextutil(8). http://www.manpagez.com/man/8/kextutil/.
[27] LLVM - Control Flow Integrity. http://clang.llvm.org/docs/ControlFlowIntegrity.html.
[28] LLVM - LibClang. https://clang.llvm.org/docs/Tooling.html.
[29] LMbench - Tools for Performance Analysis. http://www.bitmover.com/lmbench/.
[30] Machswap exploit, PsychoTea. https://github.com/PsychoTea/machswap.
[31] mach_voucher buffer overflow, Google PJ0 . https://bugs.chromium.org/p/project-zero/issues/detail?id=1004.
[32] macOS-10.12.2-Exp-via-mach_voucher, Min(Spark) Zheng. https://github.com/zhengmin1989/macOS-10.12.2-Exp-via-mach_voucher.
[33] MacOS/iOS kernel heap overflow due to lack of lower size check in getvolattrlist, Google PJ0. https://bugs.chromium.org/p/project-zero/issues/detail?id=1564.
[34] multipath kfree exploit, potmdehex. https://github.com/potmdehex/multipath_kfree.
[35] Phoenix, Siguza. https://github.com/Siguza/PhoenixNonce.
[36] Port(al) to the iOS Core, CanSecWest 2017,

Stefan Esser. https://www.slideshare.net/i0n1c/cansecwest-2017-portal-to-the-ios-core.

[37] Project zero bug reports, Google. https://bugs.chromium.org/p/project-zero/issues/list.

[38] Spice exploit, JakeBlair420. https://github.com/JakeBlair420/Spice.

[39] Supervisor mode access prevention. https://lwn.net/Articles/517475/.

[40] Sysbench. https://github.com/akopytov/sysbench.

[41] time_waste exploit for iOS 12.0-13.3 tfp0, jakeajames. https://github.com/jakeajames/time_waste.

[42] treadm1ll exploit, tihmstar. https://github.com/tihmstar/treadm1ll.

[43] Unicorn cpu emulator framework (arm, aarch64, m68k, mips, sparc, x86). https://github.com/unicorn-engine/unicorn.

[44] Uwrk - a http benchmarking tool). https://github.com/wg/wrk.

[45] v3ntex, tihmstar. https://github.com/tihmstar/v3ntex.

[46] XNU kernel heap overflow due to bad bounds checking in MPTCP, Google PJ0. https://bugs.chromium.org/p/project-zero/issues/detail?id=1558.

[47] XNU kernel UaF due to lack of locking in set_dp_control_port, Google PJ0. https://bugs.chromium.org/p/project-zero/issues/detail?id=965#c2.

[48] XNU Source Code. https://opensource.apple.com/source/xnu/.

[49] Xnu task_conversion_eval() mitigation for ikot_task ipc_object, apple. https://opensource.apple.com/source/xnu/xnu-6153.11.26/osfmk/kern/ipc_tt.c.auto.html.

[50] Xnu zone_require() mitigation for ipc_object, apple. https://opensource.apple.com/source/xnu/xnu-6153.11.26/osfmk/kern/zalloc.c.auto.html.

[51] Yalu102, qwertyoruiopz. https://github.com/kpwn/yalu102.

[52] Zerodium offering 1.5 million dollars for a Apple iOS 10 remote jailbreak. https://www.scmagazine.com/zerodium-offering-15-million-for-a-apple-\ios-10-remote-jailbreak/article/529771/.

[53] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[54] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[55] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[56] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[57] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of c++ virtual function tables in binary programs. In *Proceedings of the Computer Security Applications Conference*, 2014.

[58] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[59] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM conference on Computer and Communications Security*.

[60] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[61] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, 2015.

[62] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 2014.

[63] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[64] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security*, 2007.

[65] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[66] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[67] Laszlo Szekeres, Mathias Payer, Lenx Tao Wei, and R Sekar. Eternal war in memory. 2014.

[68] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM conference on Computer and Communications Security*, 2017.

[69] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[70] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. Vtint: Protecting virtual function tables' integrity. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, 2015.

XI. APPENDIX

Listing 7: clock_sleep_trap() system call

```
kern_return_t clock_sleep_trap(
    struct clock_sleep_trap_args *args)
{
    mach_port_name_t clock_name = args->clock_name;
    ...
    if (clock_name == MACH_PORT_NULL)
        clock = &clock_list[SYSTEM_CLOCK];
    else
        clock = port_name_to_clock(clock_name);
    ...
    if (clock != &clock_list[SYSTEM_CLOCK])
        return (KERN_FAILURE);
    ...
    return KERN_SUCCESS;
}

clock_t port_name_to_clock(
    mach_port_name_t clock_name)
{
    clock_t clock = CLOCK_NULL;
    ...
    if (ip_active(port) && \
    (ip_kotype(port) == IKOT_CLOCK))
            clock = (clock_t) port->ip_kobject;
    return (clock);
}
```

Listing 8: pid_for_task() system call

```
kern_return_t pid_for_task(
    struct pid_for_task_args *args)
{
    mach_port_name_t t = args->t;
    user_addr_t pid_addr  = args->pid;
    ...
    //get the related task
    t1 = port_name_to_task_inspect(t);
    ...
    p = get_bsdtask_info(t1);
    if (p) {
            pid  = proc_pid(p);
            err = KERN_SUCCESS;
    }
    ...
    //return the pid value to userspace
    copyout(&pid, pid_addr, sizeof(int));
    ...
    return(err);
}

task_inspect_t port_name_to_task_inspect(
    ipc_port_t port)
{
    task_inspect_t task = TASK_INSPECT_NULL;
    ...
    if (ip_active(port) && \
        ip_kotype(port) == IKOT_TASK) {
            task = port->ip_kobject;
    }
    ...
    return (task);
}

void *get_bsdtask_info(
    task_t t)
{
    return(t->bsd_info);
}

int proc_pid(
    proc_t p)
{
    if (p != NULL)
        return (p->p_pid);
    return -1;
}
```

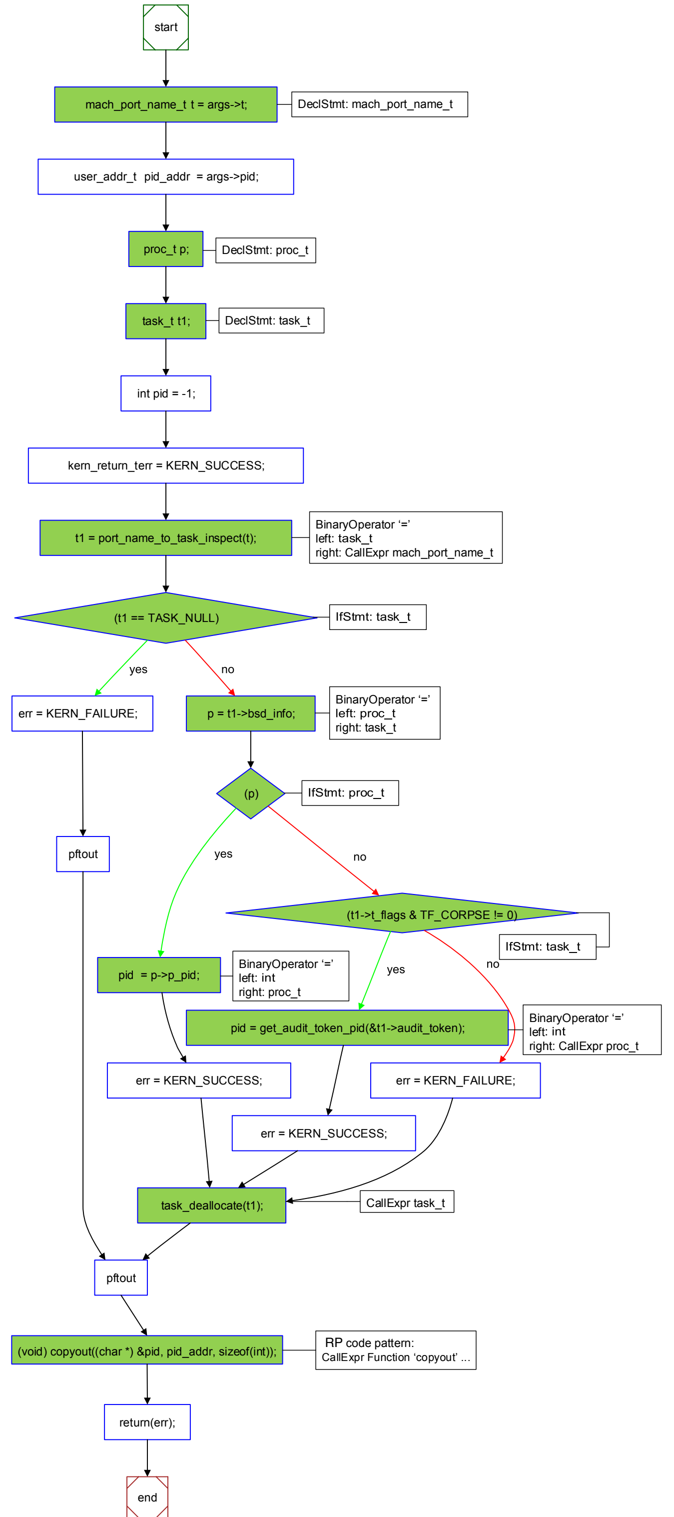

Fig. 11: (Mach) port-oriented control flow graph for pid_for_task() system call.

Listing 9: clock_get_time() system call.

```
kern_return_t clock_get_time(
    clock_t clock,
    mach_timespec_t     *cur_time)
{
    if (clock == CLOCK_NULL)
        return (KERN_INVALID_ARGUMENT);
    return ((*clock->cl_ops->c_gettime)(cur_time));
}
```

Listing 10: Mach_voucher buffer overflow.

```
kern_return_t
mach_voucher_extract_attr_recipe_trap(
    struct mach_voucher_..._args *args)
{
    ...
    mach_msg_type_number_t sz = 0;

    copyin(args->recipe_size, (void *)&sz, sizeof(sz));
    ...
    uint8_t *krecipe = kalloc((vm_size_t)sz);
    ...
    //args->recipe_size should be sz
    copyin(args->recipe, (void *)krecipe, args->recipe_size)
    ...
}
```

Listing 11: task_swap_mach_voucher reference counting.

```
kern_return_t
task_swap_mach_voucher(
    task_t          task,
    ipc_voucher_t   new_voucher,
    ipc_voucher_t   *in_out_old_voucher)
{
     if (TASK_NULL == task)
         return KERN_INVALID_TASK;

     *in_out_old_voucher = new_voucher;
     return KERN_SUCCESS;
}

mig_internal _Xtask_swap_mach_voucher(
    mach_msg_header_t *InHeadP,
    mach_msg_header_t *OutHeadP)
{
    ...
    new_voucher = convert_port_to_voucher(
                          In0P->new_voucher.name);
    old_voucher = convert_port_to_voucher(
                          In0P->old_voucher.name);
    RetCode = task_swap_mach_voucher(task,
                                    new_voucher,
                                    &old_voucher);
    ipc_voucher_release(new_voucher);
    ...
}
```