The Pennsylvania State University
The Graduate School

# INCREASING EXPLOITABILITY VIA ELASTIC KERNEL OBJECTS

A Thesis in
Informatics
by
Zhenpeng Lin



Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2021

The thesis of Zhenpeng Lin was reviewed and approved by the following:

Xinyu Xing
Assistant Professor of Information Sciences and Technology
Thesis Advisor

Linhai Song
Assistant Professor of Information Sciences and Technology
Committee Member

Ting Wang
Assistant Professor of Information Sciences and Technology
Committee Member

Mary Beth Rosson
Professor of Information Sciences and Technology
Graduate Program Director

# Abstract

Recent research endeavors have explored various methods to perform kernel exploitation and bypass kernel protection and exploitation mitigation. However, these efforts primarily focus on anecdotal methods or side-channel approaches. There have not yet been many systematic, hardware-agnostic, and general exploitation methods for exploitation mitigation circumvention. In this work, we analyze a recently released anecdotal exploit which utilizes an elastic kernel object to bypass KASLR. We hypothesize that this approach could become a general exploitation practice, through which many vulnerabilities could bypass widely deployed kernel mitigations (e.g., KASLR and heap cookie protector).

To validate our hypothesis, we design a systematic method and implement it as a semi-automated tool – EOE . By using EOE on two popular OSes (Linux and macOS), we could identify many elastic kernel objects, through which many vulnerabilities identified in their kernel code could demonstrate the ability to bypass exploitation mitigations (e.g., KASLR and heap cookie protector). For some vulnerabilities, we also show that EOE could even track down elastic kernel objects to facilitate arbitrary-read kernel exploitation. By analyzing existing kernel defense mechanisms, we argue that the newly induced exploitation method is challenging to defend. Without taking rapid action, it could inevitably become a severe threat to kernel security.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgments

Here, I would like to thank my advisor Xinyu Xing for his support, and my friends who helped me along way. This work was supported by NSF 1718459, and ONR N00014-20-1-2008. It is noted that the findings and conclusions of this paper do not necessarily reflect the view of the funding agency.

# Chapter 1
# Introduction

Over the past years, security researchers have introduced many defense mechanisms to secure the kernel and prevent it from being exploited. The techniques proposed range from the kernel protection schemes (e.g., executable space protection [1] or enforced non-executable physmaps [2]) to exploitation mitigation mechanisms (e.g., KASLR or SMEP/SMAP). Along with all these techniques comes better kernel security, and the standard exploitation methods become no longer useful. For example, with the design of KASLR, an attacker is no longer able to reliably jump to a particular exploited function in memory and thus hijack the control of the kernel.

Along with the efforts of improving kernel security, many security researchers also devote significant energies to the new methods of performing kernel exploitation to circumvent exploitation mitigation and kernel protection commonly adopted by OSes. However, many of their efforts focus on anecdotal approaches that target a specific vulnerability (e.g., [3–11]) or side-channel attacks that heavily rely upon the availability of a set of hardware features (e.g., [12–16]). To the best of our knowledge, there have not yet been many systematic, hardware-agnostic, and general exploitation approaches demonstrating effectiveness in bypassing existing kernel defenses (e.g., KASLR and heap cookie protector [17, 18]).

In this work, we manually analyze an anecdotal exploit. It demonstrates the capability of bypassing KASLR by using an out-of-bound overwrite vulnerability (CVE-2017-7184) identified in the Linux kernel. Technically, this exploit first leverages the overwriting capability to manipulate a critical field bmp_len in a kernel object xfrm_replay_state_esn. Through this practice, the attacker changes the boundary of an elastic buffer enclosed in the kernel object, tricking the kernel into authorizing him to read a memory region that he otherwise cannot be entitled to. As we elaborate in Chapter 2, by placing a pointer in the overread region referencing a global variable, the attacker could utilize a disclosure

channel copy_to_user() to uncover that pointer to the userspace and thus compute the kernel base address accordingly.

Through our analysis detailed in Chapter 2.1, we argue that the exploit mentioned above could be induced and extended as a general exploitation practice, through which an attacker could leverage an overwriting ability to manipulate an elastic kernel object and thus disclose critical kernel information. In this work, we hypothesize that, by following this general practice, most kernel vulnerabilities could potentially give an adversary the ability to bypass many kernel exploitation mitigations. Even worse, we also hypothesize that, for some vulnerabilities, an attacker can obtain the privilege to perform arbitrary read in the kernel.

To verify the hypotheses above, we have to tackle three significant challenges. Given an OS with sophisticated implementation and its update frequency, it is impractical to manually sift through millions of lines of C code and thus pinpoint elastic objects. Therefore, we first have to design an efficient, effective method to analyze kernel code and thus automatically track down elastic objects potentially useful for exploitation. Second, as is mentioned above, the general exploitation method needs to utilize a disclosure channel to pass critical information to the userspace. To do that, as we will discuss in Chapter 2, we have to ensure the existence of a channel in the kernel implementation, through which one could pass the corresponding data to the userland or a remote host. As such, we have to extend our design further to explore whether the elastic objects identified provide us with such potentials. Last but not least, we also have to augment our design with the ability to match a vulnerability to corresponding elastic kernel objects. As we will elaborate in Chapter 2, this is because vulnerabilities provide different capabilities in overwriting data, which restricts the kernel objects available for exploitation mitigation circumvention.

In this work, we explore, design, and develop a systematic method that tackles the challenges above by using static analysis as well as constraint solving. We implement this approach as a semi-automated tool and name it after EOE standing for "**E**lastic **O**bject for **E**xploitation". Using this tool for real-world vulnerabilities identified in both Linux and macOS, we show that the elastic kernel objects are pervasive in the kernel implementation across both popular OSes. For most of the vulnerabilities identified in both systems, we can always track down at least one elastic kernel object (and sometimes more), which allows us to forge a heap cookie quickly and bypass KASLR stably. With this observation and comprehensive analysis of the existing kernel defenses, we not only verify our hypothesis but, more importantly, raise the concern for the existing kernel

protection mechanisms.

To the best of our knowledge, this is the first work demonstrating elastic kernel objects could be a severe threat to many exploitation mitigation mechanisms (e.g., KALSR and heap cookie protector). As we will demonstrate by using 27 real-world kernel vulnerabilities in Chapter 6, elastic kernel objects could facilitate exploitation and bypass exploitation mitigations not only for the vulnerabilities identified on the Linux kernel but also for those tracked down on XNU. For some vulnerabilities, by using elastic objects, this work also demonstrates the capability of performing arbitrary read in the kernel. Motivated by the newly identified kernel threat, this work also analyzes existing kernel defenses, pinpoints their weakness, and eventually provides some new defense suggestions to mitigate the influence of the newly induced exploitation upon existing kernel defenses.

In summary, this paper makes the following contributions.

- We analyze an anecdotal exploit publicly released and induce its exploitation method as a general practice.
- We design and develop a systematic method, demonstrating that the general exploitation practice could facilitate most vulnerabilities identified on Linux and XNU to bypass widely deployed exploitation mitigations.
- We implement our systematic method as a semi-automated tool – EOE that facilitates the discovery of elastic kernel objects as well as their pairing with corresponding vulnerabilities.

The rest of this paper is organized as follows. Chapter 2 analyzes the anecdotal exploit and describes our research focus. Chapter 3 provides the technical overview, followed by the details in Chapter 4. Chapter 5 discusses the implementation of our proposed technique. Chapter 6 validates our hypothesis by using real-world vulnerabilities. Chapter 7 discusses the challenge of defending against the newly induced exploitation method, the possible defense, and some works relevant to ours. Finally, we conclude the work in Chapter 8.

# Chapter 2
# Exploit Analysis & Research Focus

In this chapter, we briefly describe an anecdotal kernel exploit publicly released. Motivated by this exploit, we then induce a general exploitation method and elaborate on our research focus. Finally, we discuss the threat model.

## 2.1 Anecdotal Exploit & Induced Method

**Anecdotal public exploit.** Recently, a researcher has released a working exploit that demonstrates the capability of performing out-of-bound read through a vulnerability (CVE-2017-7184) identified in Pwn2Own 2017. As is depicted in Figure 2.1, by triggering the vulnerability, the exploit first performs out-of-bound access, which overwrites the adjacent kernel object xfrm_replay_state_esn and thus enlarges the value of bmp_len. As is shown in the figure, bmp_len is a length field in xfrm_replay_state_esn. By design, it indicates the size of the buffer bmp at the end of the kernel object and controls how many bytes the system call recvmsg could read data from bmp and return to the

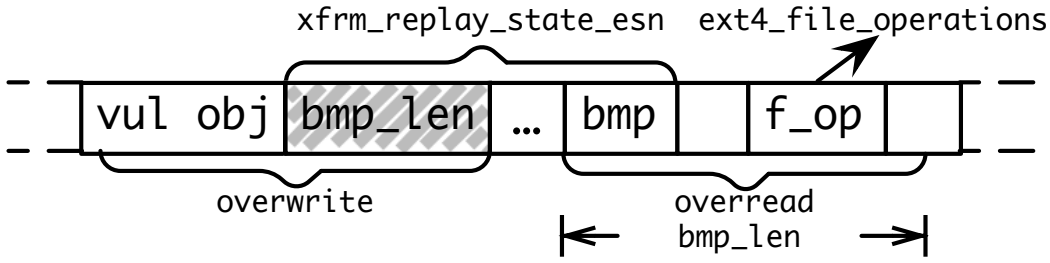


**Figure 2.1.** The illustration of the anecdotal exploit performing buffer overread and uncovering the pointer f_op referencing a global variable ext4_file_operations.

userland. As such, by overwriting bmp_len, one could naturally obtain the ability to reveal the data from the kernel object adjacent to xfrm_replay_state_esn. As is illustrated in Figure 2.1, the kernel object next to xfrm_replay_state_esn contains a pointer ext4_file_operations referencing a global variable. Through the buffer overread, the exploit could disclose that pointer, calculate the base address of kernel code, and eventually bypass KASLR.

In the exploit described above, the kernel object involved is very different from many other kernel objects, the sizes of which remain fixed. The size of the kernel object xfrm_replay_state_esn is determined by two kinds of fields – ❶ a number of fields the sizes of which are fixed (e.g., bmp_len shown in Figure 2.1) and ❷ an elastic buffer bmp, the size of which is defined by a specific field (bmp_len) in the kernel object. As such, kernel objects like xfrm_replay_state_esn are considered one kind of elastic objects. Used in the kernel development, they not only ease the implementation by minimizing the need for manually managing allocated memory [19] but, more importantly, upgrade the performance of kernel execution by improving the cache hit rate [20].

**Induced exploitation method.** Motivated by the exploit above, we hypothesize that, given any kernel vulnerability that overwrites the memory managed by the SLAB/SLUB or Zone allocator, we can potentially perform similar exploitation, disclose critical information, and thus bypass corresponding exploitation mitigations. For example, we can first perform heap manipulation and place an elastic kernel object to the target slot by using the method introduced in recent research [21, 22]. Second, by using the overwriting capability, we can manipulate the length field indicating the size of an elastic buffer. As we will discuss in Chapter 3, an elastic object could be either a kernel object that encloses the elastic buffer as part of the object or an object that contains a pointer referencing that elastic buffer outside the object. As such, the manipulation of the length field naturally provides us with the ability to overread data from different memory regions. For example, given the elastic buffer located on the stack, the manipulation of the corresponding length field would naturally provide us with the potential to overread data from the stack. Similar to the example exploit above, assuming that we could find a system call like recvmsg that could establish a channel to disclose the overread stack data to the userland, we can potentially obtain an exploitability disclosing a stack canary accordingly.

The stack canary disclosure practice discussed above is just a straightforward example demonstrating the circumvention of one specific kernel defense (i.e., stack canary). By pairing a vulnerability with elastic objects referencing elastic buffers at different memory

regions, we could go beyond stack canary circumvention. For example, if the attacker could overread a buffer on the heap and manipulate memory layout similar to what the anecdotal exploit demonstrates, he could potentially grant a vulnerability with the capability of bypassing KASLR. By following the memory manipulation method proposed in [21, 22], if the attacker could craft a memory layout, allowing the overread ability to access a freed slot, he could also potentially uncover heap cookie and thus circumvent the heap cookie protector. As we will discuss more in Chapter 4.2, for some vulnerabilities, by overwriting some elastic objects in a specific manner, an attacker can even be permitted to perform arbitrary read and thus lead to user credential steal.

## 2.2 Research Focus & Threat Model

**Research focus.** As is mentioned above, the induced exploitation approach implies a general ability to bypass many exploitation mitigation mechanisms. However, such an approach is hypothetical. In practice, we have not yet observed evidence that proves such a practice could become a general approach to facilitating the exploitation of a majority of kernel vulnerabilities. Today, the ways to circumvent the exploitation mitigation above are still heavily dependent upon anecdotal tricks [5–11], recently proposed side-channel attacks [12–16], and the nature of vulnerability [3, 4]. Our hypothetical method has not yet been considered a threat to the many kernel exploitation mitigation mechanisms (e.g., KASLR, heap cookies, and stack canary). We believe this is due to the following facts.

To employ the method summarized above, an adversary has first to ensure a kernel implementation encloses the aforementioned elastic kernel objects. However, given the complexity of kernel code and the diversity of kernel versions, it is extremely labor-intensive to track down such kernel objects by reviewing kernel code manually. Second, even if a kernel implementation relies upon elastic kernel objects, the adversary also has to guarantee the existence of a channel. Through that channel, he could pass the data stored in the elastic buffer (i.e., the buffer the size of which is indicated by one of the fields in the kernel object) back to a userland process. However, there has not yet been a systematic approach to pinpointing the link between the userland process and elastic kernel objects. Third, after identifying the elastic kernel objects with the potential to leak data to userland, it does not imply the adversary could utilize that object to leak critical kernel information. Given a vulnerability corrupting data in a particular cache/zone, an attacker cannot guarantee he could allocate his desired kernel object to the same cache. Even if both the vulnerable and elastic objects share the same cache/zone, the attacker

still needs to ensure the vulnerability gives him a sufficient ability to manipulate the length field tied to the elastic buffer.

In this work, our research focus lies in verifying our hypothetical exploitation method by tackling the challenges mentioned above. To be specific, we explore a systematic method to ❶ examining elastic objects in kernel code, ❷ evaluating their capability of leaking data to the userland process, and ❸ eventually pairing these elastic kernel objects with the capability of target vulnerabilities. Following this series of research exploration, we also analyze existing kernel defenses and investigate new defense mechanisms possibly useful for mitigating the new exploitation method.

**Threat model & assumptions.** In addition to the defense mechanisms that the hypothetical exploitation method aims to bypass, this work first assumes that kernel arms with many other exploitation mitigations and kernel protection mechanisms, such as SMEP and SMAP protection [23], KPTI protection [24], and W⊕R. These protections and mitigations are the kernel defenses most commonly enabled and adopted in both Linux and XNU. Second, we assume the kernel heap freelist has been randomized on both Linux and XNU. However, since there have already been exploitation methods [25–27] decisively bypassing kernel freelist randomization, without further clarification, this research does not consider it the obstacle of our general exploitation method. Third, it is very typical that an attacker has only one zero-day vulnerability in hand. Therefore, we do not assume the attacker has additional vulnerabilities to facilitate the exploitation and, thus, the mitigation circumvention. Finally, the capability of a vulnerability used in this work indicates at which memory addresses an adversary could overwrite data freely. It is a commonplace that an attacker obtains a vulnerability capability from a PoC program, demonstrating only kernel panic. Therefore, we conservatively assume an attacker cannot find capabilities other than that manifested through the PoC program. For example, if a PoC program overwrites only four bytes of kernel memory on the heap at the time of kernel panic, we conservatively assume the attacker could obtain only the four-byte overwriting capability.

# Chapter 3
# Overview

In the example shown in Figure 2.1, an elastic buffer is part of an elastic kernel object, placed at the end of the object. In order to identify such objects, one instinctive reaction is to search all structures with a buffer at the very end of that structure and then use off-the-shelf static analysis methods to pinpoint corresponding objects. However, this method is not likely to help us pinpoint all the elastic objects useful for exploitation and mitigation circumvention. As is detailed in the Appendix, the implementation of an elastic structure/object is very diverse. For example, a kernel developer could implement an elastic object that encloses a pointer referencing a buffer outside and then uses an enclosed length field to indicate its size. As a result, we argue that a sophisticated code analysis method is necessary for tracking down elastic objects accurately. Under the guidance of this belief, we design a systematic method below, which facilitates elastic object identification.

Based on the description above, we argue that an elastic kernel object has to enclose a length field indicating the size of a buffer regardless of where the buffer locates. In kernel implementation, a developer usually defines the length field as an integer variable. As a result, our systematic approach first examines the existence of an integer variable in each kernel structure. Then, it deems the structures with the declaration of such a variable as the candidates. As is mentioned in Chapter 2, the elastic object is used to facilitate exploitation for vulnerabilities that have the behavior of corrupting data on the heap. Therefore, the second step of our systematic approach is to examine each memory allocation site and filter out those sites at which the allocated objects are not only in types of the candidate structures but, more importantly, occupy slots on the heap. Besides, we check if these allocation sites could be reachable from at least one unprivileged system call. For each allocation site filtered out, our approach also records the size of the kernel objects allocated at that site. With this information, we can

determine the cache or zone that an elastic object potentially belongs to and, later, we can use this information to match objects with vulnerabilities accordingly.

After following the two steps mentioned above, we can exclude many kernel objects. However, this does not mean the remaining are all elastic kernel objects nor indicate they are useful for performing exploitation and bypassing mitigation. As is mentioned earlier, to be an elastic object, the length field of a kernel object has to indicate the size of a buffer, through which a userland process could read $N$ bytes of kernel data where $N$ is the number stored in the length field. As such, our method further summarizes a set of kernel functions (e.g., copy_to_user(void ___user* to, const void* from, unsigned long n) in Linux and copyout(const void* kernel_addr, user_addr_t user_addr, vm_size_t nbytes) in XNU).

For the following reasons, in this work, we deem the calling sites of the functions above as the channels through which an attacker could potentially access data stored in the elastic buffer. First, these functions are the communication channels that developers design for passing data from kernel to userland. They provide a possibility for us to migrate kernel data in the elastic buffer to the userspace. Second, these functions reference the kernel data passing to the userland by using a data pointer (e.g., const void* from in copy_to_user() and const void* kernel_addr in copyout()). They can potentially be the alias of the reference to the elastic buffer. Third, through an input argument, these functions specify the bytes of kernel data that can be copied to the userland (e.g., unsigned long n in copy_to_user() and vm_size_t nbytes in copyout()). Technically speaking, the initialization of this argument could originate from the length field in the elastic object.

With the analysis above, we further design our elastic object identification method to perform a backward taint analysis. To be specific, we take the corresponding argument of the functions above as taint source. Starting from the invocation sites of these functions, we then analyze kernel code reversely and keep checking whether the corresponding input argument originates from one of the candidate kernel objects filtered out through the first two steps mentioned above. In this work, our proposed method considers a candidate kernel object as "elastic" only if the tainted input argument is propagated to one of the candidate kernel objects. For detail of this approach, readers could refer to Chapter 4.2.

As is mentioned above, an elastic object does not directly imply exploitation facilitation nor the mitigation circumvention. First, an elastic kernel object might not be allocated to the cache/zone the same as the one holding the vulnerable object. With this situation, the overwriting capability demonstrated by that vulnerability cannot be used to manipulate

the length field and thus complete the consecutive exploitation. Second, even if an attacker could place an elastic kernel object at the same cache/zone as that of the vulnerable object, it is still unclear if the vulnerability gives him the overwriting capability of manipulating the length field freely. For example, a vulnerability may offer the overwriting ability only for the first 4 bytes of its adjacent object. However, the length field in an elastic object may start from the $5^{th}$ byte, making the exploitation infeasible. Third, at the stage of disclosing critical kernel information through system calls, a kernel execution path might impose additional data constraints, which could potentially limit the amount of kernel data that an adversary could disclose.

To tackle the problems above, as part of our systematic approach, we propose a semi-automated method. This method first manually summarizes the capability of vulnerability under the guidance of GDB [28]. Then, it automatically extracts the constraints imposed on kernel data leakage. Finally, it compares the property summary of each elastic object with the capability of vulnerability and employ a heuristic-based approach to glue elastic kernel objects with corresponding vulnerabilities. In Chapter 4.3, we describe the detail of this semi-automated solution.

# Chapter 4
# Technical Detail

In this chapter, we first elaborate on how to identify a set of elastic object candidates by following the first and second steps described above. Then, we specify how to filter out the elastic objects beneficial for bypassing exploitation mitigation. Finally, we introduce the method for pairing kernel vulnerabilities with corresponding elastic objects.

## 4.1 Identifying Elastic Object Candidates

As is mentioned in the chapter above, to identify elastic kernel object candidates, we first examine the existence of an integer variable in kernel structures. However, the definition of a structure could involve other structural variables. Therefore, to ease the integer variable identification and reduce possible mistakes, before looking for the integer field in structures, we go through each of the field members in the structure and flatten that structure as follows.

Given a structure $S$, if its field member $f_i$ is a structural variable or a nested structure, we replace $f_i$ with all its field members $[f_{i1}, f_{i2}, ..., f_{in}]$. If the field member $f_i$ is an array with more than two dimensions, we compute its total size and replace it with a single-dimensional array accordingly. If the field member $f_i$ is a union variable or a nested union, we duplicate the corresponding field member lists by copying the field members in each union and thus obtain a set of new structures $\{S_1, S_2, ..., S_u\}$ where $u$ is the number of different definitions inside the union.

In this work, we repeat the process above recursively until no more operations above can be further applied. Intuition suggests that by following the recursive procedure to pre-process a structure, each of the field members in that structure can be turned into either an ordinary data type (e.g., `char`, `int*`) or a single-dimensional array. With this, we can easily pinpoint integer variables in structures and deem a structure with an

integer field as our candidate structures.

With the candidate structures in hand, as is described in Chapter 3, the next step is to track down all the sites of heap memory allocation reachable from unprivileged system calls and further examine whether allocated objects on these sites are in the types of candidate structures. To determine the reachability from a system call to an allocation site, we check that, in between, whether there is at least one path that does not involve the kernel function call capable(CAP_SYS_ADMIN) for Linux or priv_check_cred() for XNU. The reason is that a call to either of these functions showing on all the paths toward an allocation site indicates root permission and the failure of exploitation[1]. To pinpoint the sites of heap memory allocation, we search for critical kernel functions in the kernel source code. In both Linux and XNU, there are two types of kernel functions responsible for allocating memory on the heap. One is kmalloc and kalloc series which are used for object allocation on the general cache/zone. The other is kmem−cache and mcache_alloc series which are designed for allocation on the special cache/zone.

In our design, we deem the invocation of these functions in the kernel code as the sites of memory allocation. To determine the type of objects allocated at these sites, we analyze the return values of these functions. The reasons are ❶ their return values are always pointers referencing the objects allocated, and ❷ by analyzing the type of the return value, we can easily point out whether the type of an allocated object is within the set of candidate structures. To be more specific, when analyzing the types of return values, we follow a use-def chain and resolve memory alias. As is stated in Chapter 5, we implement our use-def chain analysis by using LLVM. As a result, when performing use-def analysis, we keep track of those instructions relevant to typecasting, pointer dereferencing, and argument passing. The operands of these instructions explicitly reveal the object type. By using this information, we can easily infer and conclude the type of each allocated object accordingly.

By using the methods mentioned above, we can easily filter out all the allocated kernel objects – the types of which are within the set of candidate structures – and treat them as the candidates of elastic kernel objects. As is mentioned in Chapter 3, in order to ease the process of pairing a vulnerability with corresponding elastic kernel objects for mitigation circumvention, the second step of elastic object identification also involves the efforts of determining the cache/zone to which each of allocated objects belongs. Therefore, along with the analysis mentioned above, we also perform the analysis for

[1]Note that system calls without the root permission requirement can also be in the privilege category. However, they do not tie to the highest permission. In this work, we, therefore, treat them as unprivileged ones.

each kernel object as follows.

For allocation functions in kmem−cache or mcache_alloc series, their first argument is always a static or global pointer referencing a special cache/zone. For example, to allocate a kernel object in the type of struct seq_file, the allocation function kmem_cache_zalloc() puts seq_file_cache as its first argument, explicitly specifying the cache/zone of the object belongs to. For a kernel object allocated in this manner, we can easily point out the corresponding cache/zone in which the object fits. Different from allocation functions designed for the special cache/zone, allocation functions in kmalloc and kalloc_ series use constant or sometimes constant plus a variable as its first argument, indicating the size of the allocated object. For the functions with a constant as their first argument, we can easily associate the kernel object with the corresponding cache/zone. For example, if the Linux kernel allocates an object with 132 bytes, we can associate the cache kmalloc−192 with the object because a 132-byte kernel object is too large for kmalloc−128 and overly small for kmalloc−256. For the allocation functions with the first argument in the form of a constant plus a variable, at this particular stage, we temporarily tie the corresponding object to all the general caches/zones with the size greater than the constant.

## 4.2 Filtering out Object Candidates

Recall that an elastic kernel object also has to support data disclosure to the userspace. To further narrow down objects with such a property, we first summarize a set of critical kernel functions shown in the Appendix and deem the calling sites of these functions as the leaking anchors through which an attacker could potentially uncover kernel data to the userspace.

As is mentioned in Chapter 3, from each of the leaking anchors, we perform backward taint analysis to filter out the elastic object candidates further. As is shown in Table A.1 (presented in the Appendix), all the critical kernel functions contain two important arguments. One indicates the length of kernel data to be disclosed to the userland (e.g., the second argument attrlen in the function nla_put_nohdr()). The other specifies the address from which the kernel data would be retrieved (e.g., the last argument data in the function nla_put_nohdr()). In this work, we take both of these arguments as the taint sources and perform interprocedural backward taint analysis for each of the taint sources individually.

Starting from the taint sources indicating the length of kernel data (e.g., the argument

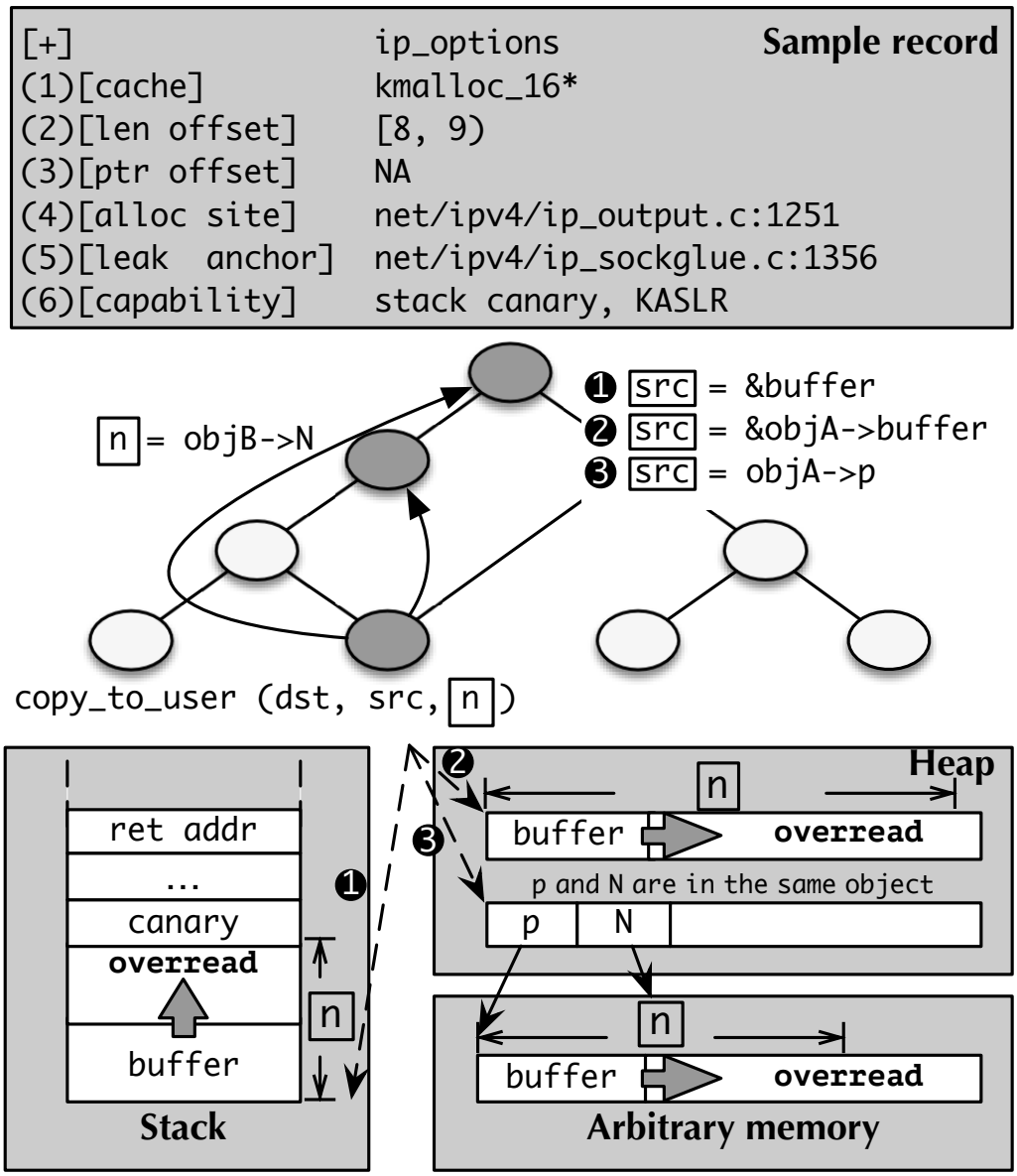


**Figure 4.1.** The illustration of backward taint analysis starting from copy_to_user(). Argument n originates from field N inside elastic object objB. From different paths, taint analysis concludes argument src could come from three different variables indicated by ❶∼❸.

n in Figure 4.1), we keep track of the data flow reversely and examine the memory regions from which these arguments originate. If the value of the length argument originates from a variable allocated on the stack or global memory region, we discard the invocation site of the corresponding kernel function because, as is mentioned earlier, our attack relies upon the power of overwriting data on the kernel heap. A length argument originating from a stack, or global variable does not hold the requirement of launching the attack successfully. For the length argument tied to a variable on the kernel heap region (e.g.,

n=objB−>N in Figure 4.1), we continue our backward taint. To be specific, we further analyze to which kernel object the variable belongs, and with which structural type the kernel object is associated.

For the calling sites preserved after the analysis above, we further perform backward analysis for the taint source corresponding to the data argument mentioned above (e.g., the argument src in Figure 4.1). However, slightly different from our backward taint analysis applied to the length argument, for taint sources tied to data arguments, we first follow the data flow reversely and track down all the sites where the corresponding data argument is initialized (e.g., ❶ src=&buffer, ❷ src=&objA−>buffer, and ❸ src=objA−>p). Starting from these variable assignment sites, we then analyze the type of the variable accordingly.

For the variable referencing a memory region on the stack (e.g., the dotted line ❶ in Figure 4.1), we conclude that an adversary could potentially obtain an ability to overread data on the stack if an overwriting capability allows the adversary to manipulate the corresponding length argument through the variable identified on the heap (e.g., objB−>N in Figure 4.1). By using this stack overread capability, we can further conclude the capability of disclosing the stack canary and the return address to bypass KASLR. Concerning the variables allocated on the non-stack region (i.e., the heap area[2]), they could be categorized into the following two types, providing an adversary with different exploitability.

The first type indicates the variable referencing the address of one field in a kernel object (e.g., the dotted line ❷ in Figure 4.1). For this type of variable, we conclude that an adversary could potentially obtain an ability to bypass KASLR or forge a legitimate heap cookie. The reason is that an adversary could utilize an overwriting capability to vary the corresponding length argument (e.g., objB−>N in Figure 4.1), follow the corresponding path to trigger the critical function (e.g., copy_to_user()), and eventually obtain an overread primitive on the heap. As is discussed earlier in Chapter 2.1, by using a state-of-the-art technique [21, 22] to manipulate heap layout, the adversary could easily turn this overread ability on the heap into the ability to bypass KASLR and heap cookie protector.

The second type of variables are pointers enclosed in the kernel objects (e.g., the dotted line ❸ in Figure 4.1). For this type of variable, before drawing any conclusion, we take one additional step, which continues backward taint analysis, and examines whether

[2]Note that the kernel needs to determine the size of the buffer on the global area at the compilation time. Therefore, a variable tied to a data argument cannot be present on the global region if the data argument references an elastic buffer.

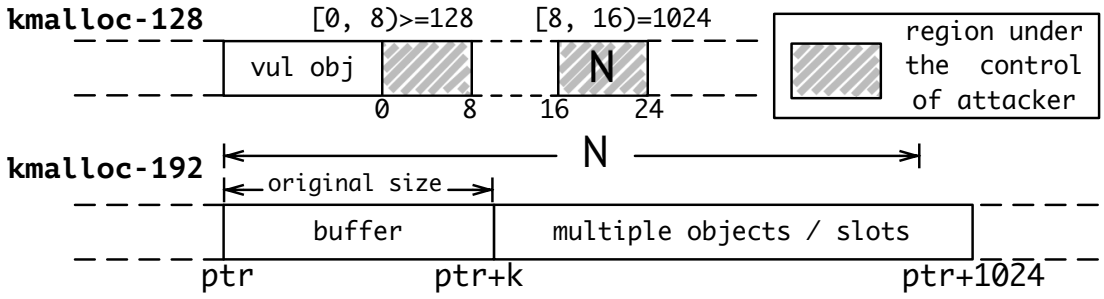


**Figure 4.2.** The summarization of vulnerability capability & demonstration of buffer overread through the capability.

the variable, as well as that tied to the length argument, are both in the same object. In this work, we conclude that the kernel objects with such a property can potentially provide an adversary with the ability to not only bypass KASLR but, more importantly, perform arbitrary kernel read. The reason behind this conclusion is as follows. For variables (associated with the length and data arguments) enclosed in the same kernel object, an adversary could potentially allocate the object in the cache/zone same as that of the vulnerable object and thus utilize the overwriting ability to manipulate both variables accordingly (e.g., manipulating p and N in Figure 4.1). Since the variable tied to the length argument indicates how many bytes of data one could read, and the other variable specifies from which memory region one could read the data, the manipulation capability turns the overwriting vulnerability into an arbitrary read primitive. With this primitive, the adversary could read the content in the interrupt descriptor table (IDT), compute the base address, and thus circumvent KASLR. Besides, as the previous research [3] has already demonstrated, the adversary could also use this arbitrary read primitive to dump memory and search for the string "root:!:" in the memory. Since this string is part of the file "/etc/shadow", the adversary could disclose a user's password easily.

In this work, for each of leaking anchors surviving from the analysis above, we store the corresponding conclusive capabilities in a database for the consecutive analysis (e.g., "stack canary" and "KASLR" depicted in Figure 4.1). For the elastic kernel objects and corresponding structures included in the candidate set but not associated with any critical functions, they do not satisfy the definition of elastic kernel objects (i.e., having a channel to disclose data in the elastic kernel buffer to userspace). Therefore, we discard such objects and structures from the candidate set. By following the analysis above as well as the way to knock off unqualified elastic objects, we can eventually filter out all the elastic kernel objects beneficial for exploitation and, thus, mitigation circumvention.

## 4.3 Pairing Vulnerabilities with Objects

Through the analysis above, we could obtain the information regarding each of the elastic objects. As is shown in Figure 4.1, the information includes (1) the caches or zones tied to each object, (2) the offset of the length field corresponding to the head of the object, (3) the offset of the elastic buffer corresponding to the object head if the pointer referring the elastic buffer and the length field share the same object, (4) the sites where the kernel allocates the object, (5) the leaking anchor(s), and (6) conclusive capability inferred through the paths toward the corresponding leaking anchor(s).

With the information in hand, given a vulnerability, we could use the following approach to pair that vulnerability with elastic objects accordingly. First, we utilize a debugging tool (GDB [28]) to track the execution of a PoC program triggering the vulnerability but not necessarily performing actual exploitation. Based on our observation from the debugging tool, we then identify the cache where the vulnerability corrupts data. Besides, we manually summarize, to which specific memory locations in that cache, the vulnerability gives an attacker the ability to overwrite data. At each location, what value range could be under an attacker's control. For example, as is shown in Figure 4.2, through our manual analysis against an out-of-bound vulnerability, we can discover that the vulnerability overflows a vulnerable object in the cache `kmalloc-128` and corrupts data in it adjacent spot. Recall that we can allocate an elastic object at that adjacent spot by using the technique proposed in [21, 22]. Therefore, we can manually summarize the region under corruption is the first and the third 8 bytes of that elastic object, and the values put into these two regions have to be greater than 128 and equal to 1024, respectively.

In this work, we deem these memorization results as the capability of a vulnerability and utilize a list of 2-tuples `[(`$VCache_1$`, `$Cap_1$`), ..., (`$VCache_n$`, `$Cap_m$`)]` to model such a capability. Here, $VCache_i$ indicates the cache the vulnerability could corrupt, and $Cap_j$ represents the range of unauthorized memory region at which vulnerability could overwrite data. Considering that, in one particular cache, a vulnerability might have the ability to modify data at multiple memory regions, we represent the $Cap_j$ as a list of assertions `[(`$R_{j1}|Op_{j1}|V_{j1}$`), ..., (`$R_{jx}|Op_{jx}|V_{jx}$`)]`. In this assertion list, the notation $R_{jk}$ indicates the unauthorized memory area where, through the corresponding vulnerability, an attacker could manipulate. The notations $Op_{jk}$ and $V_{jk}$ altogether specify the attacker's control over that region. To illustrate this, we again take, for example, the case shown in Figure 4.2. Using the representation above, we could write the capability

of that vulnerability as `(kmalloc-128, [([0, 8)⩾128), ([16, 24)=1024)])`. Here, `kmalloc-128` denotes the cache the vulnerability could corrupt. `[0, 8)⩾128` and `[8, 16)=1024` indicate the ranges of values that an adversary could put into the corresponding memory regions.

By using the modeling approach above to describe the capability of a vulnerability, we can automatically pair a vulnerability with those elastic objects useful for exploitation. To be specific, given a vulnerability, we first filter out all the elastic objects, if the caches or zones they tie to (indicated by the notation $OCache_1 \cdots OCache_h$) happen to have an overlap with the caches associated with the vulnerability (i.e., $\exists i \in [1, n],\ \exists j \in [1, h] \mid (VCache_i = OCache_j)$). For each elastic objects filtered out, we then examine whether the memory regions under manipulation cover its length field[3]. With this examination, we can preserve the elastic objects with their length field covered and thus narrow down the elastic objects useful for exploitation further. For the elastic kernel objects preserved, last but not least, we check if an adversary could use his overwriting ability to manipulate the length field and thus go over the boundary of the elastic buffer. Take the vulnerability capability mentioned above, for example. Assume the length field of an elastic object is at the third 8 bytes, and the object contains a pointer referencing a buffer in an object located at the cache `kmalloc-192`. Given part of the vulnerability capability `([16, 24)=1024)`, we can conclude the attacker could change the size of the elastic buffer to 1024 and thus overread the buffer and access the data in its entire adjacent slot (see Figure 4.2). With this ability, we can further conclude the attacker could potentially forge heap cookies and bypass KASLR because he could keep that slot adjacent to the buffer either unoccupied or occupied with an object enclosing a function pointer.

In this work, to determine the overread ability and thus conclude corresponding exploitability, we utilize the following strategies. For the elastic buffer on the heap, we check whether the newly manipulated buffer size is at least twice as large as the cache (at which the buffer is located). With this, regardless of the position of the buffer in an object, we can always guarantee to overread the entire adjacent slot or kernel object and thus potentially give an adversary the ability to bypass KASLR or forge a legitimate heap cookie. For the elastic buffer on the stack, we compute the stack frame where the elastic buffer is located and then examine whether the newly manipulated buffer size

[3]Note that we also examine the manipulable memory region covers the elastic buffer if both the pointer referring the elastic buffer and the length field share the same object. In this work, we record the elastic object with this property because this indicates the object could potentially offer the capability of arbitrary read.

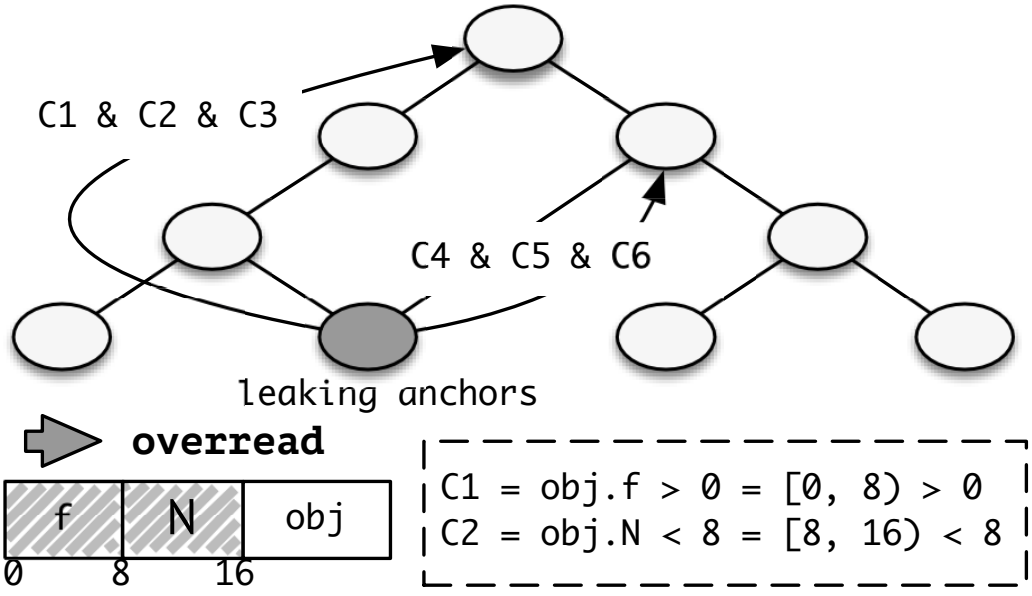


**Figure 4.3.** The illustration of constraint extraction from two paths. EOE preserves only the constraints pertaining to the manipulated fields in elastic object obj (i.e., C1 & C2).

is larger than the frame size. With this, we can ensure an adversary could always have access to the stack canary and the return address. It should be noted that the restriction we impose upon the process of determining possible exploitability is very tight. Even if some of them do not hold, it is still possible to bypass corresponding mitigation. In this work, we impose universal, tight restrictions. First, this is because the design eases our process in finding elastic kernel objects and concluding exploitability. Second, this is because the tight restriction represents the lower bound of a concluded exploitability.

Through the series of examinations above, for any individual vulnerability, we can easily track down the corresponding elastic objects friendly for exploitation. However, this does not imply that the vulnerability could automatically inherit the security implication tied to the elastic objects. On the paths toward the leakage sites after the manipulation of an elastic object, the kernel might use the manipulated fields as branch predictors. With an improper modification on some of the fields, the kernel execution might be detoured, and the kernel would no longer invoke the critical functions like copy_to_user(). In some situations, the kernel may even accidentally touch invalid or non-permitted memory regions and thus trigger general page fault (GPF) and even kernel panic. As a result, before concluding the process of pairing vulnerabilities with elastic objects, we perform further analysis as follows.

Given a vulnerability and one of its corresponding elastic objects identified through the method mentioned above, we first retrieve all its paths towards leaking anchors

(see Figure 4.3). Along each path, we first extract all the pointer dereferences. Then, we refine the set of branching constraints that must hold in order to reach out to the leakage site. For each of the dereferenced pointers, we ensure the pointer references a legit memory area if memory manipulation inevitably touches its original value. For the constraint set, we pay particular attention to the constraints in which the manipulated fields in the elastic object are enclosed or has data dependency with the variable involved. For example, as is illustrated in Figure 4.3, after the manipulation of an elastic kernel object, manipulated values fill the length field N and one of its adjacent fields f. By performing the analysis above, we can first identify two distinct paths through which an attacker could potentially disclose data through an elastic buffer. Then, we can filter out all the constraints pertaining to the length field and its adjacent field (e.g., C1=obj.f>0; C2=obj.N<8 shown in Figure 4.3). In this case, the constraints filtered out indicate the conditions that an attacker has to satisfy when crafting manipulated values for corresponding fields. In this work, we, therefore, introduce these constraints as the additional restrictions to the process of pairing vulnerabilities with corresponding elastic objects (e.g., C1&C2 in Figure 4.3).

As is depicted in Figure 4.3, similar to the way to model a vulnerability capability, we represent each manipulated field by using the offset to the head of the elastic object, and summarize the corresponding constraints in the format of `(range|op|value)`. Here, the notation `range` denotes the memory area tied to the manipulated field in the elastic object, and the notation `op|value` specifies the condition the manipulated field has to satisfy. For example, `"[8, 16)<8"` depicted in Figure 4.3 indicate that the length field locates at the second 8 bytes of the elastic kernel object. In order for being able to follow the path on the left, the value put into the length field has to be less than 8.

# Chapter 5 Implementation

In this research, we implemented our idea as a tool and named it after EOE . In the following, we present some important implementation details pertaining to the design mentioned above.

**Bitcode generation.** As is discussed in Chapter 4, our proposed method is based on static analysis. In our implementation, EOE utilizes default settings to generate LLVM bitcode files, compiling Linux and XNU kernels by using `defconfig` and `xnudeps` configuration, respectively. Then, it takes as input a list of unlinked LLVM bitcode files. During static analysis, both compilation optimization and the heavy usage of load/store instructions could increase the burden of alias analysis as well as control-flow graph construction. As a result, in order to minimize their indirect impact upon our elastic kernel object identification, improve static analysis efficiency, and reduce false negatives for elastic object identification, EOE dumps bitcode files by using WriteBitcodeToFile() provided by LLVM. To be specific, EOE invokes this method in between mem2reg pass and optimization passes, which reduces redundant load/store instructions and constructs SSA form for variables.

**Control-flow graph construction and alias analysis.** In a recent work [29, 30], researchers extend LLVM and implement a new tool for building a call graph for the Linux kernel. Technically, the tool leverages a multi-layer type analysis method[1] to construct a field-sensitive call graph. In the implementation of EOE , we borrow the call graph construction pass from this tool and use it as a building block for our kernel control-flow graph construction. To better serve our purpose, we customize the call graph by cutting off nodes and associated edges in the call graph that represent functions in . init .text section. This is because these functions cannot be invoked after kernel booting and are not useful for exploitation. Based on our customized call graph, we further

[1]The LLVM pass released on GitHub [31] has implemented only a 2-layer type analysis.

construct our context-sensitive control-flow graph. To be specific, we add intra edges between basic blocks based on the successors specified in the instruction BranchInst. Besides, we introduce inter edges by connecting Callsite to the corresponding function entry and linking ReturnInst back to the same Callsite. As is mentioned in Chapter 4.1, our proposed technique also relies upon alias analysis results to determine the type of allocated kernel objects and perform backward taint analysis. Therefore, EOE extracts alias analysis results by reusing the AliasAnalysis pass provided by LLVM project. In comparison with one-level context-sensitive Andersen's algorithm, the precision of this alias analysis is roughly the same. It should be noted that, in the process of elastic object identification, we propagate a tainted variable to its aliasing variable only if we confirm the two variables have a must-alias relationship. With this strategy, while we might introduce an under-tainting issue, this approach could significantly minimize false positives (i.e., avoiding from pinpointing the kernel objects that are not actually elastic).

**Elastic object identification.** To identify elastic object candidates, we track down memory allocation functions in kernel code. Then, we analyze the return values of these functions, determining their types. In our implementation, EOE utilizes the following three instructions to infer type information. For the instruction Getelementpr, its arguments contain a pointer referencing an object as well as the type information of that object. Through this instruction, EOE can easily obtain the type information pertaining to the object. For the instruction BitCast, one of its arguments is a pointer referencing an object. Through the other two arguments, which specify the types being cast before and after, EOE can also interpret the type information easily. With respect to the instruction CallInst, a pointer referencing an object could be enclosed as part of its arguments. Similar to the two instructions above, EOE can easily infer the type information for that object based on the type information specified along with the corresponding argument.

**Elastic object filtering.** In Chapter 4.2, we filter out the elastic object candidates by performing backward taint analysis from the length argument in the critical kernel function (e.g., the variable n in the function copy_to_user). During our analysis, the length argument can be obtained through multiple dereference. For example, in the multi-layer dereference A−>B−>len, B is the actual elastic object that encloses the length field. As a result, to avoid mistakenly taking A as the elastic object, EOE taints only one LoadInst backward if a multi-layer dereference involves at the leaking anchor. As is mentioned in Chapter 4.2, we also need to determine which memory region the corresponding function arguments refer to (e.g., stack, heap, or global). In our

implementation, EOE distinguishes memory regions by following the rules below. If the memory region is from AllocaInst instructions which allocate memory on stack, we deem the region as part of the stack. If the memory region is represented by a global/static variable, we assign it to a global area. Otherwise, we conservatively treat it as a memory region on the heap.

**Critical constraint set extraction.** As is described in Chapter 4.3, before pairing a vulnerability with elastic objects, we need to collect the constraint sets from the paths that lead kernel execution to the leaking anchor but not detour it to other sites or trigger accidental execution termination. To do this, EOE analyzes LLVM IR to determine the usage of object fields in each path towards leaking anchors based on the semantics of instructions. For example, if a field in an object is used in the instruction CmpInst, EOE first interprets the comparison by its Predicate (e.g., ICMP_ULT means the greater relationship ">"). Then, it checks which branch can reach the corresponding anchor. In this example, we keep the ">" operator if only the TRUE branch can reach the anchor. We flip the the operator into "$\leq$" if only the FALSE branch reaches the anchor or the length argument can be updated in TRUE branch. Otherwise, we discard this comparison because the corresponding constraint is not relevant to the practice of restricting kernel execution from flowing towards the leaking anchor.

**Object and vulnerability pairing.** In our implementation, EOE utilizes the `Z3` solver [32] to pair a vulnerability with elastic objects. More specifically, we use BitVec to represent the layout of an object and machine arithmetic to describe the corresponding memory range. For example, given a constraint `"[8, 16)<8"` in an object with the size of `128` bytes, we create x = BitVec('x', 128∗8) to represent the object layout, use x << (8∗8) >> (112∗8) to depict its range `[8, 16)`, and finally employ the representation (x << (8∗8) >> (112∗8) < 8) to indicate the constraint. Given an elastic object, for each path towards a leaking anchor, EOE first conjuncts the corresponding constraints on that path with a set of constraints indicating the capability of the corresponding vulnerability. Then, EOE feeds those combined constraints to the `Z3` solver. If a solution is successfully identified, it means the elastic object could be paired with the capability of the vulnerability and can be used for exploitation. Otherwise, we conclude that the elastic object under our examination cannot facilitate the exploitation of that vulnerability. It should be noted that, in the process of pairing, if one of the constraints tied to a path involves a variable (e.g., [0, 8)<cache_detail.20), EOE conservatively skips that path simply because the lack of information about that variable introduces uncertainty for using that object for exploitation. Admittedly, this implementation inevitably influences

the number of elastic objects available for exploitation. However, as we will show and discuss in Chapter 6, even with such a conservative implementation, for nearly all kernel vulnerabilities, EOE can point out at least one elastic kernel object for bypassing corresponding exploit mitigation.

# Chapter 6
# Hypothesis Validation

In this Chapter, we validate the hypothesis mentioned above. To be specific, we first introduce the setup of our hypothesis validation. Then, we elaborate on the elastic kernel objects that EOE identifies in both Linux and XNU kernels. Finally, we discuss the utility of these elastic elastic objects by using many real-world kernel vulnerabilities.

## 6.1 Hypothesis Validation Setup

**Methodology for validating elastic objects.** As is described and discussed above, our proposed techniques rely upon static analysis. However, intuition suggests it could inevitably introduce false positives (i.e., mistakenly identifying an object as an elastic one). Therefore, to ensure falsely identified objects have minimal impacts upon our hypothesis validation, we combine automated tools along with our manual efforts to examine each elastic object that EOE tracks down. To be more specific, we first instrument the Linux kernel code, placing panic functions at each leaking anchor site as well as the sites where the kernel allocates elastic objects that EOE identifies. Second, we compile the Linux kernel image that we experiment by using `defconfig` plus `KCOV` - a module that collects code coverage. Third, for each site of our interest, we perform kernel fuzzing for two hours on a machine with 2.30GHz CPU and 64GB memory by using `Syzkaller` on the Linux kernel image and manually examine every kernel object and leaking anchor that the kernel fuzzing identifies accordingly. For those elastic kernel objects and the leaking anchor that the kernel fuzzing fails to reach, we also rely upon manual efforts and examine whether these sites of our interest could be truly reachable by a concrete input. With these combined efforts of both an automated tool and manual workforce, we clean up the database that contains elastic objects, preserving only those that could be genuinely reachable through concrete user input. Regarding the XNU kernel, there are

no publicly available fuzzing tools that could ease the process of validating the legitimacy of elastic kernel objects. As a result, we solely rely upon manual efforts for this task.

**Kernel vulnerability selection.** Table B.2 (in the Appendix) lists 27 kernel vulnerabilities that we select to validate the utility of elastic kernel objects. We argue that the vulnerabilities of our selection are representative. First, as we can observe from the table, this list includes all types of vulnerabilities that corrupt data on the kernel heap (i.e., out-of-bound write, use-after-free, and double-free). Second, it covers all the heap corruption vulnerabilities used in various Linux kernel exploitation research (e.g., [21, 33–35]). Third, it encloses all XNU vulnerabilities publicly disclosed in the past three years.

It should be noted that, by publicly released XNU vulnerabilities, we mean they have to satisfy the following three criteria. ❶ A vulnerability has to demonstrate its capability through a PoC program that triggers the vulnerability but not necessarily has to perform actual exploitation. ❷ The vulnerability has to allow us to trigger it without requiring a particular hardware device nor root privilege. ❸ By running the publicly released PoC program to trigger the vulnerability, the kernel panic or failure (typically declared along with a released writeup) has to be reproducible. To ease our experiment, we migrate the Linux and XNU vulnerabilities above to one particular version of Linux kernel (v5.5.3) and one specific XNU kernel (XNU-4903.221.2), respectively. They are all the latest versions of the kernels at the time we conduct our experiment.

**Vulnerability capability summarization.** Table B.2 also summarizes the capability of each vulnerability. In order for our hypothesis validation, we extract the capability of each vulnerability from its PoC program based on the criteria below. If a vulnerability is in the type of out-of-bound write, we take its capability as the range of its overflow region and the corresponding value under its control. If a vulnerability is in the use-after-free category, we depict its capability based on how the vulnerability manipulates the freed object via the corresponding dangling pointer. If a vulnerability is in the category of double free, we treat its capability as the value under the control of the corresponding spray objects (e.g., msg_msg used in many publicly released exploits [36–38]). Upon the acceptance of this work, we will make all these vulnerabilities available in virtual machines and release the exploits crafted by using elastic kernel objects. To facilitate the assessment of this work, we have released a couple of sample exploits through an anonymously link [39]. These exploits demonstrate the capability of bypassing various kernel exploitation mitigations by using some of the elastic objects identified in this work. To the best of our knowledge, for the vulnerabilities in our initial demonstration, security

| Cache | Struct | Potential | Privilege | Constraints |
|---|---|---|---|---|
| **Linux** | | | | |
| kmalloc-8 | ipv6_opt_hdr | H | ∅ | [1, 2) < Arg |
| kmalloc-16 | ldt_struct | H & A | ∅ | [8, 12) < 65536 |
| | ip_options⋆ | anchor1: H<br>anchor2: S | ∅ | anchor1: [8, 9) < Arg<br>anchor2: [8, 9) ≠ 0 |
| kmalloc-32 | fb_info | H | NET_ADMIN | [768, 776) = kaddr |
| | ip_sf_socklist⋆ | H | ∅ | [4, 8) ≠ 0 |
| | cache_reader † | H | ∅ | [0, 8) ≠ cache_detail.20 |
| kmalloc-192 | cfg80211_scan_request⋆ | H & A | NET_ADMIN | [24, 32) ≠ null |
| **XNU** | | | | |
| mbuf | mbuf | H | ∅ | ∅ |

**Table 6.1.** Elastic kernel objects sampled from Table B.1. The "Potential" column specifies the potential that the object provides for a vulnerability. H and S indicate the potential of leaking data from the heap and stack region, respectively. A indicates the potential of performing arbitrary kernel read. In the "constraints" column, ∅ denotes data disclosure imposes no critical constraints. Arg represents a system call argument under a user's control; kaddr stands for any valid kernel address; cache_detail.20 indicates the $20^{th}$ field of the variable in the type of struct cache_detail.

researchers have not yet demonstrated the same level of exploitability.

## 6.2 Results of Identified Kernel Objects

By using EOE , we track down 61 elastic kernel objects on both Linux and XNU. Through the facilitation of the automated tool as well as our manual efforts, we confirm 53 as the true positives, which indicates the false positives introduced by our static analysis are relatively low, and the reporting results of our proposed method is trustworthy. For those falsely identified elastic objects, we further explore the root cause and discover the false positives root in the inaccurate kernel call graph construction. For example, for the kernel objects probe_resp and ctl_table, the allocation of which can only occur in the functions ieee80211_set_probe_resp() and register_leaf_sysctl_tables () when hardware devices are plugged in or at the kernel booting time, EOE mistakenly links these objects with unprivileged system calls.

For the elastic kernel objects surviving from our examination, we sample a small amount from Table B.1 shown in the Appendix and list them in Table 6.1. The results in both tables specify the kind of exploitation the elastic object could potentially facilitate. As we can observe from Table B.1, for nearly all kernel objects identified (49 out of 53), they can potentially facilitate a vulnerability to disclose data from kernel heap and thus

bypass exploitation mitigations such as KASLR and heap cookie protector. For 19 elastic kernel objects, we observe they can potentially perform arbitrary kernel read because these objects enclose both the length field and a pointer referencing the elastic buffer. For a small number of elastic objects (5 out of 53), they can provide a vulnerability with the potential to overread data from kernel stack and thus leak stack canary accordingly. By manually examining kernel code, we note that both Linux and XNU use these kernel objects frequently (with 9,291 and 6,174 sites allocating or using one of these objects in Linux and XNU, respectively). Following all these observations, we can safely conclude the elastic kernel objects and their usage are pervasive in both Linux and XNU kernels.

In Table 6.1& B.1, we also specify the privilege needed for reaching out to these objects. As we can observe, most of the elastic kernel objects (39 out of 53) require no permission for allocation and information disclosure (indicated by $\emptyset$ in the table). For the remaining kernel objects, either their allocation or consecutive information disclosure requires the privilege such as `CAP_NET_ADMIN` or `CAP_AUDIT_READ`. By default, the kernel does not grant both of these permissions to an ordinary user. However, this does not mean these objects are not helpful for exploitation because a recent research [40] has already demonstrated that an ordinary user can create a user namespace and thus naturally bypass the permission check accordingly.

In Table 6.1& B.1, we also categorize the elastic kernel objects based on the cache or zone to which they belong. As we can observe from B.1, the kernel objects identified cover most of the general and some special caches/zones (e.g., kmalloc−16384 in Linux and mbuf in XNU). For some objects that enclose the elastic buffer, their size can vary. Therefore, the kernel can allocate them to any general caches/zones with the size greater than that specified in the table. In Table 6.1& B.1, we also highlight these objects with a star symbol. These cache/zone-flexible kernel objects (17 out of 53) could significantly enrich the availability of kernel objects for exploitation and thus potentially escalate the exploitability of a vulnerability.

In Table 6.1& B.1, we finally specify the constraint set that one has to satisfy in order for disclosing kernel data to the userland successfully. As we can observe, there are only one kernel object ip_options that contains more than one set of constraints tied to different leaking anchors. It indicates that, except for this object, every elastic object discloses kernel data from only one leaking anchor. As such, as we can observe from Table 6.1, only the object ip_options provides a vulnerability with the potential to disclose data not only from the kernel heap but also from the kernel stack.

From the column "Constraints" in Table 6.1& B.1, we can also observe that there are

| CVE-ID | Capability | Suitable objects # | Security Impact |
|---|---|---|---|
| **Linux** | | | |
| 2018-6555 | kmalloc-96:[0, 8)=kaddr<br>kmalloc-96:[8, 16)=kaddr | 3 | SC, HC, BA |
| 2018-5703 | NA | 0 | NA |
| 2017-8890 | kmalloc-64:[0, 8)=kaddr:<br>[8, 16)=kaddr:[16, 18)<238:[18, 64)=* | 12 + (1) | SC, HC<br>BA, AR |
| 2017-7533 | kmalloc-96:[0, 11)=*:[11, 12)='\0' | 2 | HC, BA |
| 2017-15649 | kmalloc-4096:[2160, 2168)=* | 0 | NA |
| **XNU** | | | |
| 2019-8605 | kalloc.192:[0, 192)=* | 4 + (1) | HC, BA, AR |
| 2017-2370 | kalloc.256:[0, 256)=* | 3 | HC, BA |

**Table 6.2.** Exploitability summary sampled from Table B.2. # in the third column indicates # of elastic objects useful for the exploitation of the corresponding vulnerability. # in the parentheses indicates # of elastic objects useful for exploitation, but the paths to their leaking anchors include variables. In the last column, SC, HC, and BA signify, the vulnerability could disclose stack canary, heap cookie, and base address, respectively. AR indicates it could perform arbitrary kernel read.

a few kernel objects (marked with a dagger symbol), the constraint sets tied to which involve variables. As we specify in Chapter 5, when pairing objects with vulnerability, we conservatively discard the paths associated with these constraints and ignore the corresponding kernel objects accordingly. While this inevitably reduces the total number of elastic objects available for exploitation, their influence upon exploitability escalation is negligible because the elastic objects falling into this category are minimal (5 out of 53).

## 6.3 Results of Real-world Vulnerability

Due to the page limits, we also sample some vulnerabilities from Table B.2 shown in the Appendix and list them in Table 6.2. The results in both tables indicate the exploitability of the vulnerabilities under the facilitation of elastic kernel objects. As we can easily observe from Table B.2, about 70% (19 out of 27) vulnerabilities successfully demonstrate the ability to bypass not only KASLR but also heap cookie protector. Among these 19 vulnerabilities, 10 vulnerabilities also provide us with the ability to uncover stack canary and 6 vulnerabilities also exhibit the capability of performing arbitrary kernel

read. These observations indicate that our induced exploitation method could be viewed as a general exploitation practice, making commonly adopted exploitation mitigations futile.

As part of this hypothesis validation, we also put effort into searching publicly released exploits on the Internet extensively. To our best effort, we have not yet identified any publicly released exploits demonstrating the same level of exploitability for the vulnerabilities used for our validation. While this does not indicate these vulnerabilities initially cannot be exploited for bypassing exploitation mitigations, it implies the difficulty in doing so. As such, we argue that our induced exploitation method could be a general method to escalate the exploitability for many kernel vulnerabilities.

From Table 6.2& B.2, we can also observe that for all exploitable vulnerabilities (except for the one indicated by `CVE-2017-2370`), there are more than one elastic kernel objects useful for exploitation and mitigation circumvention. For some vulnerabilities, the number of useful kernel objects is even larger (e.g., the one indicated by `CVE-2017-7184` listed in Table B.2). From the column "Capability" in both tables, we can easily discover that this richness results from ❶ the ability to corrupt kernel heap data in various caches/zones and ❷ the ability to overwrite elastic objects with less restriction.

In this work, we argue that the richness of the elastic objects could also be very disconcerting. On the one hand, this is because more elastic objects offer more opportunities to bypass mitigations (e.g., the vulnerability tied to `CVE-2017-8890` demonstrating the ability to bypass various mitigations). On the other hand, this is because the richness potentially diversifies the way to craft a working exploit, making the pattern-based exploitation detection more challenging.

For the 8 vulnerabilities that EOE fails to pinpoint a suitable object, we perform a manual diagnosis and have the following discovery. As of the vulnerabilities tied to `CVE-2018-5703`, `CVE-2018-12233`, and `CVE-2018-1000112`, their PoC programs only demonstrate the ability to overwrite the data inside the vulnerable object. These vulnerabilities naturally fall short of the power of manipulating any fields in elastic objects. For vulnerabilities corresponding to `CVE-2018-18559`, `CVE-2017-15649`, `CVE-2017-10661`, and `CVE-2019-6225`, while their PoC programs demonstrate the ability to corrupt some data in elastic objects, EOE cannot track down any kernel object with its length field overlapping with the corrupted region. For the vulnerability indicated by `CVE-2018-4243`, although EOE identifies objects overlapping with the corrupted region, the vulnerability provides only the ability to overwrite the length field with all zeros. For all these failure cases, we have not yet discovered any public exploits demonstrating the capability of

bypassing KASLR and heap cookie or performing arbitrary kernel read. It means our induced exploitation method is at least no worse than existing exploitation knowledge.

# Chapter 7
# Discussion & Related Work

In this Chapter, we discuss the challenges of defending our induced exploitation, suggest possible defense solutions, discuss future research, and summarize the works most relevant to ours.

## 7.1 Discussion

**Defense challenges.** As is discussed in Chapter 2.1, our induced exploitation requires the manipulation of the kernel heap layout. Therefore, the existing defense most likely to mitigate our induced exploitation is heap freelist randomization [18, 41]. However, we argue that this approach cannot be an effective solution to our problem. On the one hand, this is because research [41] has already demonstrated that freelist randomization has no effects for exploiting vulnerabilities like use-after-free and double free. On the other hand, it is because there have already been many techniques proposed for bypassing this mitigation effectively (e.g., [25–27]).

In addition to memory layout manipulation, our induced exploitation needs to accurately locate and modify the length field in an elastic object. As a result, another possible existing defense is structure layout randomization [42], which shuffles the fields in a data structure at the compilation phase for preventing attackers from predicting the offset of sensitive data. In this work, we argue this defense is also not likely to be useful nor practical for our problem because it relies upon a random seed to perform randomization, and the protection of this seed is not trivial. For example, Linux distros [43] have to expose the random seed to their users for building third-party kernel modules. Besides, there are intensive on-going discussions about how to prevent a random seed from being accessed by unprivileged users on the same machine [44].

**Possible defenses.** Based on the analysis above, we believe there is a strong need to

build a new defense mechanism to hinder the general exploitation method. Here, we provide some possible defense suggestions. As part of our future work, we will explore, develop, and experiment with these possible solutions.

For example, one possible solution is to isolate identified elastic objects from other objects by creating a special cache/zone. With this isolation mechanism, an adversary will not be able to leverage the vulnerability associated with other objects to manipulate the length (and pointer) field in elastic objects. Admittedly, an elastic object itself could also be potentially vulnerable, giving attackers the ability to overwrite other elastic objects sharing the same isolated cache/zone. Given the minimal number of vulnerable elastic objects identified so far, we argue that this defense mechanism could be useful in stopping effective heap layout manipulation and thus raise the bar for the new kernel exploitation method.

In addition to isolation solution, another possible solution is to design a mechanism to enable the integrity check for the data in the length field. For example, we could first expand each of the flexible structures and introduce a checksum field. Then, when the kernel allocates the corresponding object and initializes its length field, we could encrypt it and store that encrypted value in the checksum field accordingly. With this design, at the time of accessing data from the length field for deciding the amount of data to be transferred into the userland, the kernel could easily retrieve and scrutinize the checksum. Assume the kernel protects the encryption key by leveraging hardware features (e.g., ARM PAC [45] or Intel CET [46]). We can then guarantee an adversary cannot easily forge the checksum and thus launch the attack successfully.

Last but not least, we will also explore the defense solution based on shadow memory. For example, we could build shadow memory for each of the elastic objects allocated in the kernel. In that shadow memory, we can record the actual size of the corresponding object. When the kernel discloses data at any leaking anchor, we could check whether the amount of the data migrating to the userspace is within a legitimate range. Since the construction of shadow memory inevitably introduces additional memory overhead, one of our research objectives is to explore a systematic method to minimize overhead accordingly.

## 7.2 Related Work

The works most relevant to ours include escalating exploitability for bypassing exploit mitigation and designing automated methods to facilitating exploit development. Here,

we summarize and discuss them below.

**Escalating exploitability.** In the past, many research works have been proposed to escalate exploitability for bypassing various exploit mitigation mechanisms. Among all research efforts, side-channel based methods [12, 13] are the most popular ones, which leverage hardware features to disclose critical information from the kernel. For example, by taking advantage of Intel TSX, Jang *et al.* present a highly stable timing attack against KASLR [14]. Gruss *et al.* propose to utilize pre-fetch instructions to circumvent KASLR without triggering SMEP/SMAP protection [15]. Lipp *et al.* introduce a method to read arbitrary kernel memory from userland by exploiting out-of-order execution in modern processors [16].

In addition to exploiting hardware features, researchers recently design and develop alternative methods to escalate exploitability. For example, the exploitation method `ret2dir` [35] injects exploitation payload to physmap instead of user space to circumvent SMEP/SMAP. To bypass a series of Linux kernel protection (except for KASLR), the technique `KEPLER` [34] first transforms a control-flow hijacking primitive into a stack overflow. Then, it utilizes that overflow to enable an ROP attack in the Linux kernel. To avoid being caught by CFI, Data-only attack [47] exploits the vulnerable software through corrupting data flow instead of triggering critical control flow examination.

**Facilitating exploit development.** To ease the development of an exploit, researchers have proposed many exploitation automation techniques, ranging from the works that assemble exploits fully automatically (e.g., [48–54]) to the works that partially facilitate exploit development (e.g., [55–61]). In the past, these automated techniques are primarily used for the programs in the userspace. They can barely tackle the unique challenges in the kernel. Presumably, as such, we recently witnessed many research efforts on the kernel exploitation facilitation.

For example, Xu *et al.* propose two memory collision attack mechanisms [62] to assist heap spray in kernel Use-After-Free exploitation. Lu *et al.* introduce a deterministic stack spraying exploitation method and a reliable exhaustive memory spraying technique to facilitate the exploitation of Use-Before-Initialization vulnerabilities in the Linux kernel [63]. To expedite the exploration of useful primitives for kernel Use-After-free exploitation, `FUZE` [33] searches exploitable machine states by utilizing under-context fuzzing along with symbolic execution. Chen *et al.* design a capability-guided fuzzing technique that extracts the capability for out-of-bound write vulnerabilities in the Linux kernel [64]. To obtain the desired heap layout for kernel exploitation, `SLAKE` [21] proposes a method to navigate kernel objects and then an algorithm to elastic kernel layout

automatically.

**Uniqueness of our work.** The work proposed is very different from the ones mentioned above. First, rather than exploiting hardware features, we explore exploitability escalation by exploiting the capability demonstrated by kernel vulnerabilities as well as the nature of kernel objects. Second, instead of developing yet another method to circumventing exploit mitigation such as SMEP/SMAP, we focus on the exploitation method that could bypass KASLR and heap/stack cookies or perform arbitrary read in the kernel. Third, different from the works that facilitate exploit development without considering mitigation circumvent, our proposed techniques facilitate an attacker's ability to assemble a working exploit with the capability of bypassing widely deployed kernel mitigation. Last but not least, rather than focusing on one particular type of vulnerability, this work targets exploitability escalation and exploitation facilitation for all types of vulnerabilities that could demonstrate data corruption on the kernel heap.

# Chapter 8
# Conclusion

In this work, we manually analyze an anecdotal exploit. We show that some elastic kernel objects could potentially enable the exploit mitigation circumvention in the kernel. Through a systematic study, we discover that such elastic objects are pervasive in both Linux and XNU. By using many real-world kernel vulnerabilities, we find that these elastic objects could nearly always help them bypass exploit mitigation such as KASLR, heap cookie protector, and stack canary. For some vulnerabilities, elastic kernel objects can even help them enable arbitrary kernel read and thus critical information stealth. Taking a close look at existing kernel defense mechanisms, we also discover that none of them can be useful or practical for hindering the threat imposed by elastic kernel objects. We conclude that elastic objects in the kernel could be a new threat for kernel security, and there is a new need for the security community to take rapid actions.

# Appendix A
# More Implementation Details

## A.1 Implementation of Elastic Kernel Objects

The kernel object xfrm_replay_state_esn shown in Figure 2.1 is just one kind of implementation for an elastic kernel structure/object which encloses not only a length field but also the elastic buffer. Based on our manual analysis on both Linux and XNU, we also discover three alternative implementations. Here, we summarize them below.

As is illustrated in the first example shown in Figure A.1, the first alternative implementation is to have a large buffer defined in the middle of a data object and a field within that object indicating the actual buffer size or more precisely speaking the actual bytes used for storing data. At the time of defining the actual number of bytes used for storing data, kernel typically examines the length field and ensures it does not go beyond the boundary of the large buffer. However, we discover that the kernel does not always enforce this essential check at the time of reading data from that buffer. As such, it eases an attacker's ability to manipulate the length field and thus construct a buffer overread.

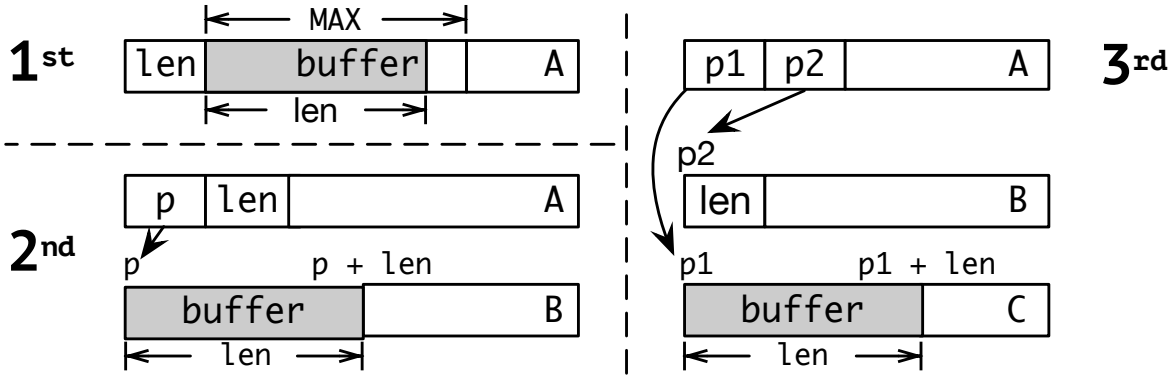


**Figure A.1.** The alternative implementations of elastic kernel objects.

| Types of Channel | Function Prototypes |
|---|---|
| User Space Memory Access APIs | unsigned long copy_to_user(void ___user* to, **const void* from**, unsigned long n); |
| | int copyout( **const void *kernel_addr**, user_addr_t user_addr, vm_size_t nbytes); |
| Netlink | int nla_put(struct sk_buff* skb, int attrtype, int attrlen, **const void* data**); |
| | int nla_put_nohdr(struct sk_buff *skb, int attrlen, **const void* data**) |
| | int nla_put_64bit(struct sk_buff *skb, int attrtype, int attrlen, **const void* data**, int padattr) |
| | void* nlmsg_data(const struct nlmsghdr* nlh); void* memcpy(void* dest, **const void* src**, size_t count); |
| | void* nla_data(const structure nlattr *nla); void* memcpy(void* dest, **const void* src**, size_t count); |
| General Networking | void* skb_put_data(struct sk_buff* skb, **const void* data**, unsigned int len); |
| | void* skb_put(struct sk_buff* skb, unsigned int len); void* memcpy(void* dest, **const void* src**, size_t count); |

**Table A.1.** The summary of the Linux/XNU critical kernel functions responsible for migrating data from kernel space to userspace. In the column of "function prototypes", the arguments in bold specify the addresses from which the kernel data originate. The arguments with wavy line indicate the amount of kernel data that an attacker can potentially disclose to the userland.

As is depicted in the second and third examples shown in Figure A.1, the rest two alternative implementations do not enclose the length field and elastic buffer in the same kernel object. Instead, they place the length field and the elastic buffer in two individual kernel objects. The difference between the second and third implementation is that one implementation contains an explicit reference to the elastic buffer and, in contrast, the other implementation references the elastic buffer through a third intermediate kernel object.

## A.2 Summary of Critical Kernel Functions

Modern OS kernels use virtual memory for isolation and provide separate address spaces for kernel and application processes. During kernel execution, however, userland processes need to inspect the on-going activities in the kernel and control kernel behavior from time to time. The kernel also needs to copy extra arguments from userland for processing and copy results back as a response to system call invocation. As a result, kernels design and implement various communication channels to facilitate data transfer between kernel space and userspace. While many communication channels (e.g., `sysfs [65]` in Linux) require root privilege or high privilege (e.g., `CAP_SYS_PTRACE`) for data migration, unprivileged users in a local or remote machine can still obtain data from kernel space through unprivileged communication channels. In this work, our induced exploitation method takes advantage of these unprivileged channels to uncover kernel data to userland processes. In the following, we summarize these communication channels, both for Linux and XNU, and categorize them into three types.

**User Space Memory Access APIs [66].** User space memory access APIs are those functions like copy_to_user() in Linux and copyout() in XNU. For copy_to_user(), this

API copies n bytes of data from kernel address from to user address to. When executing this function, the Linux kernel first ensures the destination memory region (i.e., (to, to + n]) is mapped in userspace. Then, the Linux kernel maps this region to kernel space and disables SMEP/SMAP protections (PXN/PAN on ARM) to avoid Oops or panic. For the function copyout(), it works similarly to copy_to_user(), coping nbytes data from kernel_addr to user_addr.

In addition to the two APIs mentioned above, kernels have other APIs or macros like put_user_4 to transfer data from kernel space to userspace. These APIs are similar to copy_to_user() and copyout(). However, they determine the amount of transfer data at the compilation phase (e.g., 4 bytes in put_user_4). As such, our induced exploitation method cannot manipulate such APIs. In this work, we exclude such non-manipulable APIs and list only those manipulable ones in Table A.1.

**Netlink socket family.** This channel uses the networking framework for kernel-user communication. Designed as a generic object model [67], `netlink` passes all kinds of status information about internal kernel activities to user processes (e.g., registration, removal of new devices, and hardware-related events). Although `CAP_NET_ADMIN` capability is generally required to build `netlink` socket, as is mentioned in Chapter 6, unprivileged users like containers can easily bypass this restriction by creating a user namespace (`CLONE_NEWUSER`). In Linux distributions (e.g., Ubuntu and Debian), the namespace is broadly deployed. Therefore, unprivileged users with `CAP_NET_ADMIN` capability can easily communicate with kernel through `netlink` message. For example, the anecdotal exploit discussed in Chapter 2.1 uses the channel nla_put to leak kernel data to the userspace. In Table A.1, we summarize all the kernel functions in this type. It should be noted that some communication channels are a combination of two sequential function calls.

**General Networking**. Different from the netlink socket family, which only establishes communication channels between kernel space and userspace in a local machine, APIs in general networking category enable remote data transfer. The Linux kernel sends and receives network packets by manipulating a `socket buffer`. Take the practice of sending packets for an example. In the beginning, the kernel allocates and reserves a `socket buffer` to store user data. When protocol control and user data are passed through TCP/IP layers, the kernel prepends/appends them to `socket buffer` and, at the same time, performs data validation. After this entire procedure, the NIC driver copies the entire network data in the `socket buffer` to a hardware buffer. Besides, it capsules the kernel data (e.g., authentication associated data in the link layer) into network packets.

As such, the APIs responsible for general networking operations provide adversaries with the ability to disclose data to remote hosts. In Table A.1, we list all the functions in this type and specify the number of data that the kernel can capsule into network packets.

# Appendix B
# More Evaluation Details

## B.1 More Details about Hypothesis Validation

**Elastic object identification.** Table B.1 lists all the elastic kernel objects that we identify and confirm on both Linux and XNU kernels. To be specific, the results in the table (from the column on the left to the right) indicate (1) the caches/zones to which each elastic object belongs, (2) the structural type of each elastic object, (3) the offset of the length field in each elastic object and, if applicable, that of the pointer referencing the elastic buffer, (4) the security capabilities that each object could potentially provide, (5) the privilege needed for using each elastic object to perform exploitation, (6) the constraints that an adversary has to satisfy in order for using each object for disclosing critical kernel data successfully.

In Table B.1, for clarification, we mark all the special caches/zones with a triangle symbol $\triangle$ and highlight with a star symbol $\star$ the objects that could belong to all the caches/zones greater than they are specified in the table. Besides, we utilize the symbol $\emptyset$ to signify no privilege is required if an attacker performs exploitation with the corresponding object. Similarly, the same symbol $\emptyset$ in the constraint column indicates no restriction is imposed on the manipulated elastic object. In the struct column, we use the dagger symbol † to indicate the objects the constraint sets of which involve variables.

For the mathematical notations depicted in the constraint column, Arg signifies the argument of a system call, the value of which is under the attacker's control. p$\neq$null indicates that a pointer p referencing the elastic object should not equal to a null value. The notations compat_getdents_callback.3, msgrcv_nocancel_args.7, and cache_detail.20 all represent variables, the values of which are undecidable through static analysis. For example, cache_detail.20 indicates the $20^{th}$ field of the object in the structual type cache_detail. The notation kaddr represents a valid kernel address. The formula

| Cache | Struct | Offset (len/ptr) | Potential | Privilege | Constraints |
|---|---|---|---|---|---|
| **Linux** | | | | | |
| kmalloc-8 | ipv6_opt_hdr | [1, 2)/NA | H | ∅ | [1, 2) < Arg , p ≠ null |
| kmalloc-16 | sock_fprog_kern | [0, 2)/[2, 10) | H & A | NET_RAW | [0, 2) ≤ Arg |
| | policy_load_memory | [0, 4)/[4, 12) | H & A | ∅ | [0, 4) < Arg |
| | ldt_struct | [8, 12)/[0, 8) | H & A | ∅ | [8, 12) < 65536 |
| | ip_options⋆ | [8, 9)/NA | anchor1: H<br>anchor2: S | ∅ | [8, 9) < Arg, anchor1 in put_cmsg()<br>[8, 9) ≠ 0, anchor2 in do_ip_getsockopt() |
| kmalloc-32 | cfg80211_pkt_pattern | [16, 20)/[8, 16) | H & A | NET_ADMIN | ∅ |
| | user_key_payload⋆ | [16, 18)/NA | H | ∅ | [16, 18) < Arg |
| | xfrm_replay_state_esn⋆ | [0, 4)/NA | H | NET_ADMIN | ∅ |
| | ip_sf_socklist⋆ | [4, 8)/NA | H | ∅ | [4, 8) < Arg, [4, 8) ≠ 0 |
| | cache_reader † | [24, 28)/NA | H | ∅ | [0, 8) ≠ cache_detail.20 |
| | tc_cookie | [8, 12)/[0, 8) | H & A | NET_ADMIN | [8, 12) ≠ 0 |
| | cfg80211_bss_ies⋆ | [24, 28)/NA | H | NET_ADMIN | [24, 28) ≠ 0, p ≠ null |
| kmalloc-64 | sg_header | [4, 8)/NA | H | ∅ | ∅ |
| | inotify_event_info | [36, 40)/NA | H | ∅ | ∅ |
| | fb_cmap_user | [4, 8)/[8, 16), [16, 24), [24, 32) | S | ∅ | [4, 8) ≠ 0 |
| | cache_request | [40, 44)/[32, 40) | H & A | ∅ | [20, 24) ≠ 0, [40, 44) ≠ 0 |
| | msg_msg | [24, 32)/[32, 40) | H & A | ∅ | [24, 32) < Arg, [24, 32) ≤ 4048 |
| | fname⋆ † | [44, 45)/NA | H | ∅ | [44, 45) ≤ compat_getdents_callback.3,<br>p ≠ null, [32, 40) == null, [40, 44) < Arg |
| | ieee80211_mgd_auth_data⋆ | [48, 52)/NA | H | ∅ | ∅ |
| | tcp_fastopen_context | [32, 36)/NA | S | ∅ | [32, 36) < Arg |
| kmalloc-96 | request_key_auth | [48, 52)/[40, 48) | H & A | ∅ | [48, 52) < Arg, p ≠ null |
| | xfrm_algo_auth⋆ | [64, 68)/NA | H | NET_ADMIN | ∅ |
| | cfg80211_wowlan_tcp | [28, 32)/[32, 40), [56, 62)/NA,<br>[80, 84)/[84, 88) | H & A | NET_ADMIN | ∅ |
| | xfrm_algo⋆ | [64, 68)/NA | H | NET_ADMIN | ∅ |
| | xfrm_algo_aead⋆ | [64, 68)/NA | H | NET_ADMIN | ∅ |
| kmalloc-192 | cfg80211_scan_request | [32, 36)/[24, 32) | H & A | NET_ADMIN | p ≠ null, [24, 32) ≠ null |
| | mon_reader_bin | [16, 20)/[24, 32) | H & A | ∅ | [16, 20) < 4096, [16, 20) ≠ 0, [16, 20) < Arg, |
| | cfg80211_sched_scan_request | [40, 44)/[32, 40) | H & A | NET_ADMIN | [8, 16) == kaddr, [48, 56) == kaddr, p ≠ null |
| kmalloc-256 | mon_reader_text | [112, 116)/[116, 124) | H & A | ∅ | [112, 116) < Arg |
| | station_info | [120, 124)/[112, 120) | H & A | NET_ADMIN | ∅ |
| kmalloc-512 | ext4_dir_entry_2⋆† | [6, 7)/NA | H | ∅ | [6, 7) ≤ compat_getdents_callback.3 |
| | xfrm_policy | [372, 373)/NA | S | NET_ADMIN | ∅ |
| kmalloc-1024 | fb_info | [816, 824)/[808, 816) | H & A | ∅ | [832, 836) == 0, [768, 776) == kaddr |
| kmalloc-2048 | audit_rule_data⋆ | [1036, 1040)/NA | S | AUDIT_CONTROL<br>AUDIT_READ | ∅ |
| kmalloc-16384 | n_tty_data | [8800, 8804)/NA | H | ∅ | [8800, 8804) < 4096 |
| proc_dir_entry_cache △ | proc_dir_entry † | [177, 178)/NA | H | ∅ | p ≠ null, [177, 178) ≤ compat_getdents_callback.3 |
| seq_file_cache △ | seq_file † | [24, 28)/NA | H | ∅ | [24, 28) ≠ 0, [24, 28) < Arg, [24, 28) < seq_file.1,<br>[96, 104) == kaddr |
| **XNU** | | | | | |
| kalloc.16 | user_ldt | [4, 8)/NA | H | ∅ | [4, 8) ≤ 8192, [4, 8) ≤ Arg |
| | sockaddr⋆ | [0, 1)/NA | H | ∅ | [0, 1) ≤ 255 |
| | accessx_descriptor⋆ | [0, 4)/NA | H | ∅ | [0, 4) ≠ 0 |
| kalloc.32 | msg † | [16, 18)/NA | H | ∅ | [16, 18) < msgrcv_nocancel_args.7, [16, 18) > 0 |
| | audit_sdev_entry | [8, 16)/[0, 8) | H & A | ∅ | ∅ |
| kalloc.64 | uio⋆ | [16, 24)/NA | H | ∅ | [16, 24) ≤ 4096 |
| | user_msghdr_x | [40, 44)/[32, 40) | H & A | ∅ | [40, 44) ≠ 0 |
| kalloc.80 | kauth_filesec⋆ | [36, 40)/NA | H | ∅ | ∅ |
| | vm_map_copy | [16, 24)/NA | H | ∅ | [16, 24) ≠ 0 |
| kalloc.192 | nfsbuf | [112, 116)/[136, 144) | H & A | ∅ | [112, 116)>0 |
| kalloc.224 | audit_sdev | [140, 144)/NA | H | ∅ | ∅ |
| kalloc.1024 | necp_client⋆ | [800, 808)/NA | H | ∅ | [800, 808) < Arg |
| mbuf | mbuf | [24, 28)/NA | H | ∅ | ∅ |
| pipe_zone | pipe | [0, 4)/[16, 24) | H & A | ∅ | [104, 108) ≠ 0 |
| NFS.mount | nfsmount | [196, 200) | H | ∅ | ∅ |
| mecache.necp.flow | necp_client_flow | [120, 128)/[128, 136) | H | ∅ | [120, 128) ≠ 0, [128, 136) ≠ null |

**Table B.1.** Elastic kernel objects identified and confirmed. For a detailed explanation of the listed results, readers could refer to the corresponding text in the Appendix. For the discussion of the results, readers should refer to the text in Chapter 6.

`[768,776]== kaddr`, for example, indicates the value at the memory range `[768,776]` has to be a valid kernel address.

Last but not least, in the potential column, we use three different characters to represent the potential capability of an elastic object. The characters H and S indicate the object could potentially allow an adversary to leak data from heap and stack, respectively. The character A denotes the potential of performing an arbitrary kernel read.

| CVE-ID | Type | Capability | Suitable objects # | Security Impact |
|---|---|---|---|---|
| **Linux** | | | | |
| 2018-6555 | UAF | kmalloc-96:[0, 8)=kaddr, kmalloc-96:[8, 16)=kaddr | 3 | SC, HC, BA |
| 2018-5703 | OOB | NA | 0 | NA |
| 2018-18559 | UAF | kmalloc-2048:[1328, 1336)=* | 0 | NA |
| 2018-12233 | OOB | NA | 0 | NA |
| 2017-8890 | DF | kmalloc-64:[0, 8)=kaddr:[8, 16)=kaddr:[16, 18)<46:[18, 64)=* | 12 + (1) | SC, HC, BA, AR |
| 2017-7533 | OOB | kmalloc-96:[0, 11)=*:[11, 12)='\0' | 2 | HC, BA |
| 2017-7308 | OOB | kmalloc-1024:[0, 1024)=*, kmalloc-2048:[0, 2048)=* | 12 + (1) | SC, HC, BA |
| 2017-7184 | OOB | kmalloc-32:[0, 32)=*, kmalloc-64:[0, 64)=*, kmalloc-96:[0, 96)=*, kmalloc-128:[0, 128)=* kmalloc-196:[0, 192)=*, kmalloc-256:[0, 256)=*, kmalloc-512:[0, 512)=* | 22 + (2) | SC, HC, BA, AR |
| 2017-6074 | DF | kmalloc-256:[0, 8)=kaddr:[8, 16)=kaddr:[16, 18)<238:[18, 256)=* | 11 + (1) | SC, HC, BA |
| 2017-2636 | DF | kmalloc-8192:[0, 8)=kaddr:[8, 16)=kaddr:[16, 18)<8174:[18, 8192)=* | 10 + (1) | HC, BA |
| 2017-17053 | DF | kmalloc-16:[0, 8)=* | 3 | SC, HC, BA |
| 2017-17052 | UAF | kmalloc-256:[0, 8)=kaddr, kmalloc-256:[8, 16)=kaddr | 3 | SC, HC, BA |
| 2017-15649 | UAF | kmalloc-4096:[2160, 2168)=* | 0 | NA |
| 2017-10661 | UAF | kmalloc-256:[192, 200)=kaddr, kmalloc-256:[200, 208)=kaddr | 0 | NA |
| 2017-1000112 | OOB | NA | 0 | NA |
| 2016-6187 | OOB | kmalloc-8:[0, 8)=*, kmalloc-16:[0, 16)=*, kmalloc-32:[0, 32)=* kmalloc-64:[0, 64)=*, kmalloc-128:[0, 128)=* | 23 + (2) | SC, HC, BA, AR |
| 2016-4557 | UAF | kmalloc-256:[56, 64)=*, kmalloc-256:[64, 72)=* | 3 | HC, BA |
| 2016-10150 | UAF | kmalloc-64:[24, 32)=*, kmalloc-64:[32, 40)=* | 3 | SC, HC, BA, AR |
| 2016-0728 | UAF | kmalloc-256:[0, 8)=* | 2 | HC, BA |
| 2014-2851 | UAF | kmalloc-192:[0, 8)=* | 2 | HC, BA |
| 2010-2959 | OOB | kmalloc-256:[0, 256)=* | 11 + (1) | SC, HC, BA |
| **XNU** | | | | |
| 2019-8605 | UAF | kalloc.192:[0, 192)=* | 4 + (1) | HC, BA, AR |
| 2019-6225 | UAF | kalloc.96:[8, 16)=* | 0 | NA |
| 2018-4243 | OOB | kalloc.16:[0, 8)=0 | 0 | NA |
| 2018-4241 | OOB | kalloc.2048:[0, 2048)=* | 5 | HC, BA |
| 2017-2370 | OOB | kalloc.256:[0, 256)=* | 3 | HC, BA |
| 2017-13861 | DF | kalloc.192:[0, 192)=* | 4 + (1) | HC, BA, AR |

**Table B.2.** The exploitability summary of kernel vulnerabilities. For a detailed explanation of the listed results, readers could refer to the corresponding text in the Appendix. For the discussion of the results, readers should refer to the text in Chapter 6.

**Kernel vulnerabilities and their exploitability.** Table B.2 lists all the kernel vulnerabilities used for validating our hypothesis. From the column on the left to the right, the results shown in the table indicates (1) the CVE-ID associated with the kernel vulnerability, (2) the vulnerability type into which the vulnerability was categorized, (3) the capability of the vulnerability summarized manually, (4) the total number of elastic kernel objects useful for the exploitation of the vulnerability, (5) the security implication tied to the exploitation.

In the capability column of Table B.2, as is specified in Chapter 4.3, the capability of each vulnerability is represented in the format of `cache`:[`range|operator|value`, $\cdots$]. For example, the formula `kmalloc-96:[0,8)=kaddr` indicates the vulnerability offers the ability to manipulate the first byte of an elastic kernel object, and the manipulated value is a valid kernel address. As we can observe from the table, in addition to using the notation *kaddr* to denote a valid kernel address, we introduce the symbol $*$ indicating an arbitrary value. For example, `kmalloc-2048:[1328,1336)=*` signifies the vulnerability allows an attacker to assign an arbitrary value to the memory region `[1328,1336)`.

In the column marked with "Suitable object #", we specify the total number of elastic kernel objects useful for exploitation and mitigation circumvention. It should be noted that the number without parentheses denotes the total amount of objects associated with constraints involving no variables. The number with parentheses indicates the amount of those associated with constraints involving variables. As we discuss in Chapter 5, when pairing a vulnerability with elastic objects, EOE ignores the paths involving variables and discard corresponding kernel objects (if for that elastic object EOE identifies no other paths without variable involvement). This conservative design inevitably reduces the elastic objects available for exploitation. However, as we can observe in Table B.2, even without using objects in this type, vulnerabilities can still find alternative objects for performing successful exploitation.

As is shown in Table B.2, the vulnerabilities selected for hypothesis validation cover all types such as out-of-bound write (OOB), use-after-free (UAF), and double free (DF). In the last column of the table, for each vulnerability, we also specify all their security impacts. The notations we use for indicating these impacts are SC, HC, BA, and AR. They denote the capabilities of performing arbitrary kernel read (AR) as well as bypassing stack canary (SC), heap cookie protector (HC), and leaking base address or, in other words, KASLR (BA).

# Bibliography


[1] Team, P. (2000), "Design & implementation of PAGEEXEC," .

[2] Corbet, J. (2016), "Exclusive page-frame ownership," https://lwn.net/Articles/700647/.

[3] Konovalov, A. (2018), "A proof-of-concept exploit for CVE-2017-18344," https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-18344/poc.c.

[4] Salls, C. (2017), "Exploiting CVE-2017-5123 with full protections. SMEP, SMAP, and the Chrome Sandbox!" https://salls.github.io/Linux-Kernel-CVE-2017-5123/.

[5] Lexfo (2018), "CVE-2017-11176: A step-by-step Linux Kernel exploitation," https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html.

[6] 0x3f97 (2018), "cve-2017-8890 root case analysis," https://0x3f97.github.io/exploit/2018/08/13/cve-2017-8890-root-case-analysis/.

[7] Disclosure, S. S. (2017), "SSD Advisory – Linux Kernel AF_PACKET Use-After-Free," https://ssd-disclosure.com/archives/3484.

[8] Konovalov, A. (2017), "A proof-of-concept local root exploit for CVE-2017-6074," https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-6074/poc.c.

[9] Popov, A. (2017), "CVE-2017-2636: exploit the race condition in the n_hdlc Linux kernel driver bypassing SMEP," https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html.

[10] Konovalov, A. (2017), "Exploiting the Linux kernel via packet sockets," https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[11] Horn, J. (2018), "A cache invalidation bug in Linux memory management," https://googleprojectzero.blogspot.com/2018/09/a-cache-invalidation-bug-in-linux.html.

[12] Hund, R., C. Willems, and T. Holz (2013) "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*.

[13] Evtyushkin, D., D. Ponomarev, and N. Abu-Ghazaleh (2016) "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[14] Jang, Y., S. Lee, and T. Kim (2016) "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[15] Gruss, D., C. Maurice, and A. Fogh (2016) "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[16] Lipp, M., M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg (2018) "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*.

[17] Cook, K. (2017), "mm: add SLUB free list pointer obfuscation," `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2482ddec`.

[18] Esser, S. (2016), "iOS 10 - Kernel Heap Revisited," .

[19] dp304 (2018), "Alternative to flexible array members for avoiding multiple allocations," `https://tinyurl.com/3pca77w8`.

[20] Spudd86 (2010), "Flexible array member in C-structure," `https://stackoverflow.com/questions/3047530/flexible-array-member-in-c-structure`.

[21] Chen, Y. and X. Xing (2019) "SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[22] Chen, Y., X. Xing, and J. Su (2019), "Hands off and putting SLAB/SLUB fengshui in a blackbox," `https://i.blackhat.com/eu-19/Wednesday/eu-19-Chen-Hands-Off-And-Putting-SLAB-SLUB-Feng-Shui-In-A-Blackbox.pdf`.

[23] Corbet, J. (2012), "Supervisor mode access prevention," `https://lwn.net/Articles/517475/`.

[24] ——— (2017), "The current state of kernel page-table isolation," `https://lwn.net/Articles/741878/`.

[25] Labs, A. (2020), "Grooming the iOS Kernel Heap," https://azeria-labs.com/grooming-the-ios-kernel-heap/.

[26] Hautebas, M. (2018), "empty_list - exploit for p0 issue 1564 (CVE-2018-4243) iOS 11.0 - 11.3.1 kernel r/w," https://github.com/Jailbreaks/empty_list.

[27] Beer, I. (2017), "Exception-oriented exploitation on iOS," https://googleprojectzero.blogspot.com/2017/04/exception-oriented-exploitation-on-ios.html.

[28] Stallman, R. M. (2019), "GNU Debugger," https://www.gnu.org/software/gdb/.

[29] Lu, K. and H. Hu (2019) "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[30] Lu, K., A. Pakki, and Q. Wu (2019) "Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*.

[31] Kengiter and adityapakki (2019), "Crix: Detecting Missing-Check Bugs in OS Kernels," https://github.com/umnsec/crix.

[32] Research, M. (2020), "Z3," https://github.com/Z3Prover/z3.

[33] Wu, W., Y. Chen, J. Xu, X. Xing, W. Zou, and X. Gong (2018) "FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*.

[34] Wu, W., Y. Chen, X. Xing, and W. Zou (2019) "KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*.

[35] Kemerlis, V. P., M. Polychronakis, and A. D. Keromytis (2014) "ret2dir: Rethinking Kernel Isolation," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*.

[36] Nikolenko, V. (2016), "CVE-2016-6187: Exploiting Linux kernel heap off-by-one," https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit.

[37] ——— (2018), "Dissecting a 17-year-old kernel bug," https://duasynt.com/slides/bevx-talk.pdf.

[38] ——— (2018), "Linux Kernel universal heap spray," https://duasynt.com/blog/linux-kernel-heap-spray.

[39] Authors, A. (2020), "Sample exploits," https://www.dropbox.com/sh/llj17vzdv3apmx5/AAC61uX1EPMhpTO_b7UQds8sa?dl=0.

[40] Chiang, E. (2019), "User Namespaces," https://ericchiang.github.io/post/user-namespaces/.

[41] Cook, K. (2017), "security things in Linux v4.14," https://outflux.net/blog/archives/2017/11/14/security-things-in-linux-v4-14/.

[42] ——— (2017), "security things in Linux v4.13," https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/.

[43] Hussein, N. (2017), "Randomizing structure layout," https://lwn.net/Articles/722293/.

[44] Horn, J. (2020), "Linux Email list: CONFIG_DEBUG_INFO_BTF and CONFIG_GCC_PLUGIN_RANDSTRUCT," https://www.spinics.net/lists/bpf/msg16648.html.

[45] Corbet, J. (2017), "ARM pointer authentication," https://lwn.net/Articles/718888/.

[46] Lu, H. (2018), "Intel CET - Control-flow Enforcement Technology," https://www.linuxplumbersconf.org/event/2/contributions/147/attachments/72/83/CET-LPC-2018.pdf.

[47] Chen, S., J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer (2005) "Non-Control-Data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*.

[48] Brumley, D., P. Poosankam, D. X. Song, and J. Zheng (2008) "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*.

[49] Avgerinos, T., S. K. Cha, B. L. T. Hao, and D. Brumley (2011) "AEG: Automatic Exploit Generation," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*.

[50] Cha, S. K., T. Avgerinos, A. Rebert, and D. Brumley (2012) "Unleashing Mayhem on Binary Code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*.

[51] Shoshitaishvili, Y., R. Wang, C. Hauser, C. Kruegel, and G. Vigna (2015) "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*.

[52] Shoshitaishvili, Y., R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna (2016) "SoK:(State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*.

[53] Stephens, N., J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna (2016) "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*.

[54] Hu, H., Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang (2015) "Automatic Generation of Data-oriented Exploits," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*.

[55] Bao, T., R. Wang, Y. Shoshitaishvili, and D. Brumley (2017) "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*.

[56] Repel, D., J. Kinder, and L. Cavallaro (2017) "Modular Synthesis of Heap Exploits," in *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*.

[57] Heelan, S., T. Melham, and D. Kroening (2018) "Automatic Heap Layout Manipulation for Exploitation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*.

[58] ——— (2019) "Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[59] Wang, Y., C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou (2018) "Revery: From Proof-of-Concept to Exploitable," in *Proceedings of the 25nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[60] Ispoglou, K. K., B. AlBassam, T. Jaeger, and M. Payer (2018) "Block Oriented Programming: Automating Data-Only Attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[61] Yun, I., D. Kapil, and T. Kim (2020) "Automatic Techniques to Systematically Discover New Heap Exploitation Primitives," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*.

[62] Xu, W., J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu (2015) "From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[63] Lu, K., M. Walter, D. Pfaff, and S. Nürnberger and Wenke Lee and Michael Backes (2017) "Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*.

[64] Chen, W., X. Zou, G. Li, , and Z. Qian (2020) "KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*.

[65] Mochel, P. and M. Murphy (2020), "sysfs - The filesystem for exporting kernel objects," `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`.

[66] Jones, M. (2010), "User space memory access from the Linux kernel," `https://developer.ibm.com/technologies/linux/articles/l-kernel-memory-access/`.

[67] Mauerer, W. (2008), "Professional Linux Kernel Architectures," Chapter 12.11.