A Systematic Study of Elastic Objects in Kernel Exploitation

Yueqi Chen
ychen@ist.psu.edu
The Pennsylvania State University

Zhenpeng Lin
zplin@psu.edu
The Pennsylvania State University

Xinyu Xing
xxing@ist.psu.edu
The Pennsylvania State University

ABSTRACT

Recent research has proposed various methods to perform kernel exploitation and bypass kernel protection. For example, security researchers have demonstrated an exploitation method that utilizes the characteristic of elastic kernel objects to bypass KASLR, disclose stack/heap cookies, and even perform arbitrary read in the kernel. While this exploitation method is considered a commonly adopted approach to disclosing critical kernel information, there is no evidence indicating a strong need for developing a new defense mechanism to limit this exploitation method. It is because the effectiveness of this exploitation method is demonstrated only on anecdotal kernel vulnerabilities. It is unclear whether such a method is useful for a majority of kernel vulnerabilities.

To answer this question, we propose a systematic approach. It utilizes static/dynamic analysis methods to pinpoint elastic kernel objects and then employs constraint solving to pair them to corresponding kernel vulnerabilities. In this work, we implement our proposed method as a tool - `ELOISE`. Using this tool on three popular OSes (Linux, FreeBSD, and XNU), we discover that elastic objects are pervasive in general caches. Evaluating the effectiveness of these elastic objects on 40 kernel vulnerabilities across three OSes, we observe that they can enable most of the vulnerabilities to bypass KASLR and heap cookie protector. Besides, we also observe that these elastic objects can even escalate the exploitability of some vulnerabilities allowing them to perform arbitrary read in the kernel. Motivated by these observations, we further introduce a new defense mechanism to mitigate the threat of elastic kernel objects. We prototype our defense mechanism on Linux, showing this mechanism introduces negligible overhead.

CCS CONCEPTS

• **Security and privacy** → *Operating systems security*; *Software security engineering*.

KEYWORDS

OS Security; Vulnerability Exploitation

ACM Reference Format: Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A Systematic Study of Elastic Objects in Kernel Exploitation. In *roceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3372297.3423353

1 INTRODUCTION

Over the past years, security researchers have introduced many defense mechanisms to harden the kernel, preventing it from being exploited (e.g., [18, 58, 77]). Under the protection of these techniques, common exploitation methods are no longer useful. For example, the design of KASLR no longer allows an attacker to hijack the control flow of the kernel and thus reliably jump to a particular exploited function in memory.

Responding to the effort of kernel defense development, security researchers recently devote significant energy to developing methods to circumvent exploitation mitigation and kernel protection commonly adopted by OSes (e.g., [3, 20, 24, 27, 31, 34, 37, 41–43, 45, 46, 57, 61]). Among all these efforts, one commonly adopted approach is to leverage an overwriting primitive to manipulate an elastic kernel object and thus bypass KASLR. Technically, this method first leverages an overwriting capability to manipulate a length field in a kernel object. The length field indicates the boundary of an elastic buffer enclosed in the kernel object. By manipulating this field, the attacker can trick the kernel into authorizing him/her to read a memory region that he/she otherwise cannot be entitled to. As we elaborate in Section 2, by placing a pointer in the overread region referencing a global variable, the attacker could utilize a disclosure channel to uncover that pointer to the userspace and compute the kernel base address accordingly.

In the past, security researchers have utilized anecdotal kernel vulnerabilities to demonstrate the effectiveness of this exploitation practice in bypassing KASLR. They even show that this method can be extended, potentially helping an attacker disclose a stack-/heap cookie and even perform arbitrary read. However, by far, it is unclear whether this exploitation approach is useful for a majority of kernel vulnerabilities[1]. As such, we have no clue whether this method should raise our serious concern and motivate us to develop a new kernel defense to mitigate the threat of elastic kernel objects.

To answer this question, one instinctive reaction is to demonstrate exploitability by manually crafting exploits of many kernel vulnerabilities. However, given the sophistication of the kernel code, this approach inevitably introduces a significant amount of manual effort, limiting the possibility of scaling this approach to various OSes. Moreover, given the complexity of kernel exploitation, the conclusion drawn through this manual approach might heavily rely upon the expertise of security researchers.

In this work, we design and develop a systematic method to explore the effectiveness of the exploitation method mentioned above. Our basic idea is to utilize static/dynamic analysis to identify

[1]Note that without further clarification, the kernel vulnerabilities we refer to are those that corrupt (or, in other words, manipulate) data on a heap area. The vulnerabilities with only a read capability are excluded.

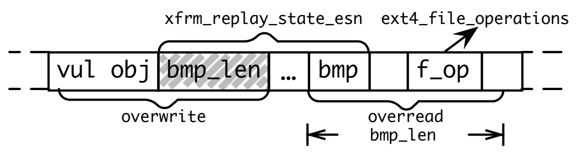


Figure 1: The illustration of the anecdotal exploit performing buffer overread and uncovering the function pointer `f_op` referencing the kernel function `ext4_file_operations`.

elastic kernel objects and then employ constraint solving to pair elastic objects with corresponding kernel vulnerabilities. We implement this approach as a tool and name it after ELOISE standing for "**E**xploitab**L**e **O**bject d**IS**cov**E**ry". Using this tool, we show that elastic kernel objects are pervasive in the kernel implementation across three popular OSes (Linux, XNU, and FreeBSD). For many vulnerabilities identified in these OSes, our tool could track down at least one elastic kernel object (and sometimes more), which allows an attacker to disclose heap/stack cookie, bypass KASLR, or perform arbitrary read. Motivated by this observation, we further introduce a new defense mechanism to mitigate the threat of elastic kernel objects. Our basic idea is to isolate elastic kernel objects in independent caches. In this way, most of the kernel vulnerabilities are no longer able to manipulate the data in an elastic object and thus trick the kernel into disclosing the critical information to the userspace.

In summary, this paper makes the following contributions.

- We design and develop a systematic method, demonstrating that a commonly adopted exploitation method is a severe threat to existing kernel defenses. It allows a majority of kernel vulnerabilities to disclose heap/stack cookie or bypass KASLR (27 out of 40). Besides, it enables some vulnerabilities to perform arbitrary read in the kernel (8 out of 40).
- We implement our systematic method as a tool – ELOISE that facilitates the discovery of elastic kernel objects and their pairing with corresponding vulnerabilities. A user study shows that the tool could significantly expedite the development of kernel exploits.
- We design and prototype a new defense mechanism on Linux to mitigate the threat of elastic kernel objects. Experiments show that the new defense significantly mitigates the threat of elastic objects and, more importantly, introduces negligible overhead to an OS (0.19%).

2 BACKGROUND

In this section, we define elastic kernel objects. Then, we briefly describe how to use these objects to perform exploitation and thus bypass kernel mitigations, followed by the challenges of this exploitation method. Finally, we discuss the threat model.

What is an elastic object? An elastic object always contains a length field that controls the size of an elastic kernel buffer. When the kernel access the data in the buffer, the length field indicates the range of the data that the kernel can read or write. As is summarized in the Appendix A.1, the implementation of an elastic structure/object is very diverse. For example, an elastic object could be a kernel object that encloses the elastic buffer as part of the object (see the 1st in Figure 5) or an object that contains a pointer referencing a buffer outside the object (see the 2nd in Figure 5). Using elastic objects, the kernel developers could minimize their need for manually managing allocated memory [21] and, more importantly, upgrade the performance of kernel execution by improving the cache hit rate [64].

How to use an elastic object to bypass exploit mitigation? We use a real-world example to illustrate how to leverage an elastic object to perform kernel exploitation and bypass mitigation. As is depicted in Figure 1, `xfrm_replay_state_esn` is an elastic kernel object that contains an elastic buffer `bmp` at the end of the kernel object. `bmp_len` is a length field that controls how many bytes the system call `recvmsg` could read data from `bmp` and return to the userland. To perform exploitation, an attacker could utilize the overwriting ability from the vulnerability to enlarge the value of `bmp_len` and thus obtain the ability to disclose the data in `bmp` buffer and the kernel object adjacent to `xfrm_replay_state_esn`. As is illustrated in Figure 1, the kernel object next to `xfrm_replay_state_esn` contains a function pointer referencing `ext4_file_operations`. Through the buffer overread, the attacker could disclose the address of the function, calculate the base address of kernel code, and eventually bypass KASLR.

In addition to bypassing KASLR, an elastic object can also facilitate the disclosure of stack/heap cookies and even enable arbitrary read. For example, if the elastic buffer is located on the stack, an overread of this buffer can cause access to unauthorized data on the stack (such as stack cookie). If the buffer is located on the heap and its adjacent slot is in free status, the overread could unveil the freed slot's metadata and thus leak the encoded heap cookie accordingly. For some vulnerabilities, an attacker can tamper the value of the pointer arbitrarily. In this case, the attacker can access nearly any memory addresses, and an arbitrary read can be easily granted.

Challenges of using elastic objects for kernel exploitation. To perform the exploitation described above, an adversary first has to ensure a kernel implementation uses the 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 auditing kernel code manually. Second, even if a kernel implementation relies upon elastic kernel objects, the adversary also has to guarantee the existence of leaking channel. Through this channel, he/she could pass the data stored in the elastic buffer (i.e., the buffer the size of which is indicated by the length fields in the kernel object) back to a userland process. However, there has not yet been a systematic approach to pinpointing the link between elastic kernel objects and the userland process. 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/she 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.

Threat model & assumptions. In addition to the defense mechanisms that the exploitation method aims to bypass, this work first assumes that the kernel is armed with other exploitation mitigations and kernel protection mechanisms, such as SMEP and SMAP protection [17], KPTI protection [19], and W⊕R. These protections and mitigations are the kernel defenses most commonly enabled and adopted in FreeBSD, Linux, and XNU. Second, we assume the kernel heap freelist has been randomized on both Linux and XNU (FreeBSD has no such protection). However, since there have already been exploitation methods [6, 28, 44] decisively bypassing kernel freelist randomization, without further clarification, this research does not consider it the obstacle of the 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 region an adversary could overwrite data freely. Since an attacker can obtain a vulnerability capability from a PoC program which only panics kernel, 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.

3 TECHNICAL APPROACH

To tackle the challenges mentioned above, we first introduce a method to identify a set of elastic object candidates in the kernel. Then, we specify how to filter out the elastic objects useful for bypassing exploitation mitigation. Finally, we introduce the method for pairing kernel vulnerabilities with corresponding elastic objects.

3.1 Identifying Elastic Object Candidates

Tracking down elastic structure candidates. Recall that an elastic object has to contain a length field. As a result, we first examine the existence of an integer variable in kernel structures. Due to the complexity of kernel implementation, the definition of a structure could involve other structural variables. 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, 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.

Pinpointing elastic object candidates at allocation sites. With the candidate structures in hand, our next step is to first track down all the sites of heap memory allocation. Then, we examine whether allocated objects at these sites are in the types of candidate structures. Finally, we examine whether reaching these allocation sites requires any root privilege. In this work, we preserve all the objects that survive the checks above, and treat them as our elastic object candidates.

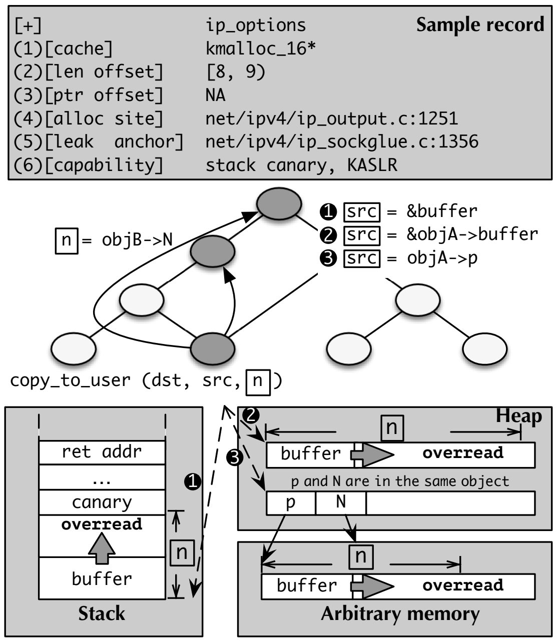
To pinpoint the sites of heap memory allocation, we search for critical kernel functions in the kernel source code. In Linux, FreeBSD, and XNU, there are two types of kernel functions responsible for allocating memory on the heap. One is `kmalloc`, `kalloc`, and `malloc` series which are used for object allocation on the general cache/zone. The other is `kmem-cache`, `mcache_alloc` and `uma_zalloc` 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 Section 4, 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 type casting, 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.

To determine the privilege required to allocate a kernel object, 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, `priv_check`, `priv_check_cred` for FreeBSD, and `priv_check_cred()` for XNU. The reason is that a call to any of these functions on the paths toward an allocation site indicates root permission and the failure of exploitation[2]. Besides, we also check whether any of the callee functions along the path towards an allocation site accesses a device that only privileged users can visit. The reason behind this is that access to privileged devices implies a high privilege for corresponding object allocation. In this work, we examine the function's access to privileged devices by using device operator structure (e.g., `struct cdevsw`). Such a structure contains a function pointer through which we can obtain the access control list of the corresponding devices. If the device's permission is root, we will exclude corresponding allocation sites and candidate objects.

Profiling elastic object candidates. Recall that our proposed method includes a component that pairs a vulnerability with corresponding elastic kernel objects for mitigation circumvention. To enable this component, as we will discuss in Section 3.3, we need to know the property of elastic objects (e.g., the cache/zone to which an elastic object belongs). As a result, we further perform analysis and profile kernel object as follows.

[2]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.



```
[+]                  ip_options              Sample record
(1)[cache]           kmalloc_16*
(2)[len offset]      [8, 9)
(3)[ptr offset]      NA
(4)[alloc site]      net/ipv4/ip_output.c:1251
(5)[leak  anchor]    net/ipv4/ip_sockglue.c:1356
(6)[capability]      stack canary, KASLR
```

Figure 2: 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.

For allocation functions in `kmem-cache`, `mcache_alloc`, or `uma_zalloc` ↪ 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 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`, `kalloc`, and `malloc` 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.

3.2 Filtering out Object Candidates

Recall that the length field in an elastic object indicates the size of a kernel buffer. Also, to use an elastic object to perform exploitation, one must ensure there is a channel to disclose the data in the buffer to the userspace. However, the object candidates identified above do not imply that their enclosed integer variable represents the size of a kernel buffer nor that they support data disclosure. As a result, we further narrow down objects with such properties. To do this, we summarize a set of critical kernel functions (Table 3 in the Appendix) and deem the calling sites of these functions as the leaking anchors through which an attacker could potentially uncover data in a kernel buffer to the userspace. With these leaking anchors in hand, we then perform a backward data flow analysis, filtering out the candidate objects that satisfy the properties mentioned above.

As is shown in Table 3 (presented in the Appendix), all the critical kernel functions contain two important parameters. 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 `n` in Figure 2), 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, the success of the exploitation 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 2), we preserve the calling sites and further perform backward analysis for the taint source corresponding to the data argument mentioned above (e.g., the argument `src` in Figure 2). Slightly different from our backward taint analysis applied to the length argument, 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 2), 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 2). 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[3]), 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 2). 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`

[3]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.

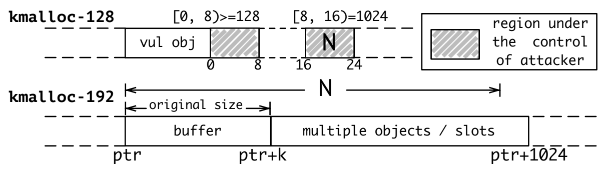


Figure 3: The summarization of vulnerability capability & demonstration of buffer overread through the capability.

in Figure 2), follow the corresponding path to trigger the critical function (e.g., `copy_to_user()`), and eventually obtain an overread primitive on the heap. Using a state-of-the-art technique [11, 12] 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 2). For this type of variable, before drawing any conclusion, we take one additional step, which continues backward taint analysis, and examines whether 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 2). 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 [43] 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 hashed password and potentially recover the password by using password cracker (e.g., John the Ripper password cracker [56]).

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 2). 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 useful for exploitation and, thus, mitigation circumvention.

3.3 Pairing Vulnerabilities with Objects

Through the analysis above, we could obtain the information regarding each of the elastic objects. As is shown in Figure 2, 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 [65]) 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 3, 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 [11, 12]. 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 summarized 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 3. Using the representation above, we could write the capability of that vulnerability as `(kmalloc-128, [([0, 8)⩾128), ([8, 16)=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[4]. 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 3). With this ability, we can further conclude the attacker could potentially forge heap cookies and bypass KASLR because he/she 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 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, it is because the design eases our process in finding elastic kernel objects and concluding exploitability. Second, it 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). 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, 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). 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).

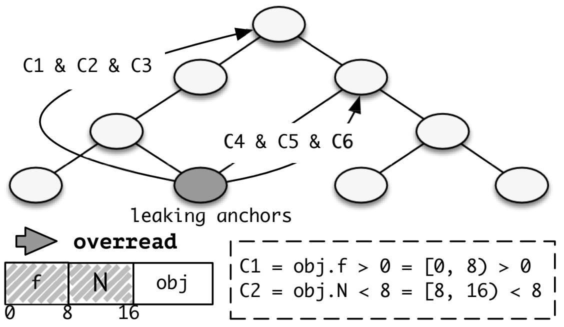


Figure 4: The illustration of constraint extraction from two paths. `ELOISE` preserves only the constraints pertaining to the manipulated fields in elastic object `obj` (i.e., `C1 & C2`).

As is depicted in Figure 4, 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 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.

4 IMPLEMENTATION

In this research, we implemented our idea as a tool and named it after `ELOISE`. In the following, we present some important implementation details pertaining to the design mentioned above. We release our code and exploits at [2] to foster future work.

Bitcode generation. As is discussed in Section 3, our proposed method is based on static analysis. In our implementation, `ELOISE` utilizes default settings to generate LLVM bitcode files, compiling

[4]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.

Linux, FreeBSD, and XNU kernels by using defconfig, GENERIC 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, ELOISE dumps bitcode files by using WriteBitcodeToFile() provided by LLVM. To be specific, ELOISE 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 [47, 48], 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[5] to construct a field-sensitive call graph. In the implementation of ELOISE, 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 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 Section 3.1, our proposed technique also relies upon alias analysis results to determine the type of allocated kernel objects and perform backward taint analysis. Therefore, ELOISE 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, ELOISE 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, ELOISE 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, ELOISE 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, ELOISE can easily infer the type information for that object based on the type information specified along with the corresponding argument.

[5]The LLVM pass released on GitHub [40] has implemented only a 2-layer type analysis.

Elastic object filtering. In Section 3.2, 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(void __user* to, const void* from, ↪ unsigned long n)). 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, ELOISE taints only one LoadInst backward if a multi-layer dereference involves at the leaking anchor. As is mentioned in Section 3.2, we also need to determine which memory region the corresponding function arguments refer to (e.g., stack, heap, or global). In our implementation, ELOISE 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 Section 3.3, 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, ELOISE 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, ELOISE 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 "≤" 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, ELOISE utilizes the Z3 solver [60] 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, ELOISE first conjuncts the corresponding constraints on that path with a set of constraints indicating the capability of the corresponding vulnerability. Then, ELOISE 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), ELOISE 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 Section 5, even with such a conservative implementation, for nearly all kernel vulnerabilities, ELOISE can point out at least one elastic kernel object for bypassing corresponding exploit mitigation.

5 EVALUATION

We design three experiments to evaluate the effect of elastic objects on kernel protection circumvention across three open-sourced kernels (Linux, FreeBSD, and XNU).

5.1 Experiment Design

Experiment I: evaluating our tool and characterizing elastic objects. As is described and discussed above, ELOISE relies upon static analysis. However, the accuracy of static analysis heavily relies upon the correctness of the kernel's control flow graph and the level of optimization introduced at the compilation stage. As a result, ELOISE inevitably introduces false positives (i.e., mistakenly identifying an object as an elastic one) and false negatives (i.e., failing to pinpoint an elastic object or successfully identifying an elastic object but failing to associate it with a disclosure channel). In this work, we design an experiment to evaluate the false positives (FP) and false negatives (FN).

To evaluate false negatives, ideally, we should follow a two-step procedure. First, we go through all the object allocation sites in the kernel implementation and determine whether the type of the allocated objects is one of the elastic structures defined in Section 3.1. Second, for every elastic object confirmed in the first step, we extract its corresponding elastic buffer and then examine whether the kernel can pass the data in the elastic buffer to the userland through the disclosure functions defined in Table 3 (e.g., copy_to_user() or copyout()). After each step, we compare the results obtained through the manual efforts with those reported by ELOISE and thus pinpoint the false negatives accordingly.

Given the complexity and huge codebase of kernel implementation, this manual approach, however, is infeasible. Take Linux kernel as an example. Its implementation (v5.5.3) contains about 14 million lines of C code, 53,775 allocation sites, and 75,100 sites that invokes one of the kernel functions listed in Table 3. Even after successfully confirming an allocated object is in the type of an elastic structure, a kernel expert still needs to manually walk through tens of thousands of C code lines to determine whether the allocated object has a connection with the corresponding disclosure kernel functions. As such, we study false negatives through a random sampling approach.

First, we randomly sampled 800 out of 6,383 elastic structures (Note that these structures have no FN for sure). Then, we manually identify all of their allocation sites in Linux, followed by checking their connection with the disclosure functions. In this work, we identify false negatives by comparing our manually analyzed results with those of ELOISE. We argue this false-negative measure is representative even if we do not perform this analysis on all the allocation sites nor apply this manual effort on other kernels (i.e., FreeBSD and XNU). It is because ❶ our allocation site sampling is random, covering 12.5% of the kernel allocation sites; ❷ the design of three kernels shares many common properties, and the false negatives observed on one kernel could be potentially generalized to other kernels; ❸ our manual analysis relies upon the workforce of three Linux kernel researchers who have extensive experience in reporting Linux kernel bugs and publishing kernel research at top tier system and security conferences.

To evaluate false positives, we leverage automated tools along with our manual effort. For Linux and FreeBSD kernels, we first instrument panic functions at the sites where ELOISE identifies the allocation of elastic objects, and the sites where ELOISE discovers the disclosure of the data from an elastic buffer. Then, for each instrumented site, we use Syzkaller[25] to assist us in checking whether there is indeed a concrete input that could reach these sites to allocate elastic objects and thus disclose data in the elastic buffers. Following the kernel fuzzing, we manually examine the sites which the fuzzing fails to reach because these unreachable sites could result from either our inaccurate static analysis or the limitation of our fuzzing technique. In this work, we take the unreachable cases as the false positives of ELOISE only if our manual review still cannot find a concrete input to reach out to our interest sites. It should be noted that there are no fuzzing tools publicly available for the XNU kernel. As a result, we solely rely upon manual effort for the false-positive study on XNU. For all the true positives, we then conduct further experiment to understand the pervasiveness of elastic objects and study their characteristics.

Experiment II: evaluating the ability to bypass mitigation. Recall that one of our objectives is to study whether elastic kernel objects could enable many kernel vulnerabilities to bypass widely deployed exploit mitigation methods, such as KASLR, heap protector, and stack canary. To answer this question, we select 40 kernel vulnerabilities and summarize the capability of each kernel vulnerability. Then, we use ELOISE to examine whether one of the identified elastic objects could be paired with these vulnerabilities and hence enable kernel protection bypassing.

Due to the space limit, we list these 40 vulnerabilities in Table 7 and describe how we manually summarize vulnerability capability in the Appendix. We argue that the selected vulnerabilities are representative. First, 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., [11, 39, 79, 80]). Third, it includes 10 Linux kernel vulnerabilities randomly sampled from syzbot [26] (a Linux kernel vulnerability dashboard). These randomly sampled vulnerabilities are identified by Syzkaller recently but have not yet assigned with an CVE ID. Finally, it encloses all XNU and FreeBSD vulnerabilities publicly disclosed in the past three years.

It should be noted that, although the National Vulnerability Database lists many CVEs relevant to FreeBSD and XNU, a majority of them cannot be used for our evaluation because the detail of the vulnerabilities is incomplete or completely missing or the trigger of the vulnerabilities requires a particular hardware. As such, we chose our FreeBSD and XNU test cases by 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, FreeBSD, and XNU vulnerabilities above to one particular version of Linux kernel (v5.5.3), one specific FreeBSD (v12.1), 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.

Experiment III: evaluating the utility of ELOISE in exploit development assistance. To evaluate the effectiveness of ELOISE in expediting exploit development, we conduct a user study under IRB permission (#STUDY00010080). In particular, we recruit our subjects from a pool of CTF players and a pool of security analysts who have extensive experience in debugging Linux kernel and writing kernel patches. Our primary recruitment method is to invite participants through emails. In our invitation email, we describe our study objective as a paid task that quantitatively measures the time spent on exploit development and qualitatively surveys the difficulty in writing an exploit. In our email, we also require all the participants to ❶ have experience in exploiting Linux kernel vulnerabilities, ❷ be familiar with the commonly adopted exploitation method discussed in this paper, and ❸ be comfortable for conducting this study online. To evaluate whether applicants qualify for this study and form groups, we asked all applicants to sign the participation agreement and complete a self-assessment form (see Figure 6 in Appendix A.5). The collected self-assessment forms indicate that, among the 8 participants, 6 subjects meet with our requirements. Regarding the expertise level, 4 subjects have about three years of experience in exploiting kernel vulnerabilities (high expertise), and the rest 2 subjects have about one year of the corresponding experience (moderate expertise). Based on these responses, we randomly assign each group with 2 highly skillful subjects and 1 subject with moderate experience. From the highly skillful subjects, we randomly picked one from each group as the leader to fill in the short survey form during the experiment.

We only include Linux kernel vulnerabilities in the experiment because the Linux kernel is fully open-sourced and thus we avoid introducing the reverse engineering skillset which is noise to our experiment. While conducting the user study, it is possible that some participants have already established the prior knowledge for some Linux kernel vulnerabilities or already known some public exploits that leverage elastic objects to bypass KASLR. As a result, to prevent the measurement bias introduced by these prior knowledge, we design our kernel vulnerability sets without those kernel vulnerabilities that already covered by the participants' knowledge base (Question 6 in self-assessment form, Figure 6). In the the first row of Table 9, we list the 5 vulnerabilities satisfying our selection criteria.

For both groups, we first gave the 5 vulnerabilities and their corresponding PoCs that trigger the vulnerabilities but not perform exploitation. Then, we asked both groups to develop exploits to bypass KASLR by using the exploitation method mentioned in this paper. We provided the kernel objects with function pointer for the two groups to help them leak the base address. For Group A, we also equipped them with our tool ELOISE, which assists them in searching elastic objects and pairing vulnerabilities with objects accordingly.

Cache	Struct	Potential	Privilege	Constraints
		FreeBSD		
kmem.32	i40e_nvm_access	H	∅	[12, 16) < 4097
		Linux		
kmalloc-8	ipv6_opt_hdr	H	∅	[1, 2) < Arg
kmalloc-16	ldt_struct	H & A	∅	[8, 12) < 65536
	ip_options⋆	anchor1: H anchor2: S	∅	anchor1: [8, 9) < Arg 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 1: Elastic kernel objects sampled from Table 4~6. 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`.

In this experiment, we kept track of the performance of two groups through short surveys every 30 minutes and thus evaluated the utility of ELOISE both quantitatively and qualitatively. As is shown in Figure 7, the survey partitions an exploit development into 3 critical stages – ❶ vulnerability capability exploration, ❷ elastic object identification, ❸ memory layout manipulation. While receiving the survey, the participants need to specify the stage of their exploit development and report their progress at that stage. This short survey could be completed in one minute without intervening the exploitation development too much. If completing one stage of exploit development, the participants also need to turn in their results. From the response to the survey inquiry, we roughly measured the time spent at each stage and thus quantify the utility of ELOISE. At the end of the user study, we also handed out a post-test survey (see Figure 8) through which we qualitatively evaluate the utility of ELOISE and understand the challenges in exploit development. Due to the space limit, we leave the three survey forms in Appendix A.5.

5.2 Results of Experiment I

Falsely identified & missing elastic objects (FP/FN). By using ELOISE, we track down 97 elastic kernel objects tied to 98 disclosure functions on Linux, FreeBSD, and XNU. We compare these objects and disclosure functions with those randomly sampled and manually confirmed. We find our manually audited objects and disclosure functions are a subset of those pinpointed by ELOISE. This discovery cannot directly conclude zero false negatives because the kernel's scale limits our ability to manually audit all objects. However, it implies the false negatives of ELOISE are minimal.

For 97 kernel objects that ELOISE identifies, we confirm 74 as the true positives, which indicates the false positives are moderate, and the reporting results of our proposed method is valuable. 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, ELOISE mistakenly links these objects with unprivileged system calls.

Objects' exploitability & pervasiveness. For the elastic kernel objects surviving from our examination (i.e., true positives), we sample a small amount from Table 4∼6 and list them in Table 1. The results in both tables specify the kind of exploitation the elastic object could potentially facilitate. As we can observe from Table 4∼6, for nearly all kernel objects identified (70 out of 74), 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 28 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 74), 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 Linux, FreeBSD, and XNU use these kernel objects widely (with 39,483, 44,956, and 22,307 sites allocating or using one of these objects in Linux, FreeBSD, and XNU, respectively). Following all these observations, we safely conclude the elastic kernel objects and their usage are pervasive in three different kernels.

Elastic objects that require high privilege. In Table 1, 4, 5 and 6, we also specify the privilege needed for reaching out to these objects. As we can observe, most of the elastic kernel objects (60 out of 74) 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 [13] has already demonstrated that an ordinary user can create a user namespace and thus naturally bypass the permission check accordingly.

Objects' cache/zone coverage. In Table 1, 4, 5 and 6, we also categorize the elastic kernel objects based on the cache or zone to which they belong. As we can observe from Table 4∼6, the identified objects cover most of the general and some special caches/zones (e.g., `kmalloc-16384` in Linux, `mbuf` in FreeBSD, and `pipe_zone` in XNU). For some objects that enclose the elastic buffer, their size can vary. Therefore, the kernel can allocate them to general caches/zones with the size greater than that specified in the table. In Table 1, 4, 5 and 6, we also highlight these objects with a star symbol. These cache/zone-flexible kernel objects (18 out of 74) could significantly enrich the availability of kernel objects for exploitation and thus potentially escalate the exploitability of a vulnerability.

Constraints tied to objects & their security implication. In Table 1, 4, 5, and 6, 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 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.

CVE-ID or Syzkaller-ID	Capability	Suitable objects #	Security Impact
	FreeBSD		
2016-1887	zone_mbuf:[0, 256)=*	1	BA, AR [6]
	Linux		
bf96...[74]	ip_dst_cache:[64, 68)=*	0	NA
2018-6555	kmalloc-96:[0, 8)=kaddr kmalloc-96:[8, 16)=kaddr	3	SC, HC, BA
2018-5703	NA	0	NA
2017-8890	kmalloc-64:[0, 8)=kaddr: [8, 16)=kaddr:[16, 18)<238:[18, 64)=*	12 + (1)	SC, HC 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 2: Exploitability summary sampled from Table 7. # 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.

From the column "Constraints" in Table 1, 4, 5 and 6, we can also observe that there are a few kernel objects (marked with a dagger symbol), the constraint sets of which involve variables. As we specify in Section 4, 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 exploit mitigation bypassing is negligible because the elastic objects falling into this category are minimal (10 out of 74).

5.3 Results of Experiment II

Summary of effectiveness in bypassing mitigation. Due to the page limit, we only sample some vulnerabilities from Table 7 and show them in Table 2. The results in both tables indicate the exploitability of the vulnerabilities under the facilitation of elastic kernel objects. As we can observe from Table 7, about 67.5% (27 out of 40) vulnerabilities successfully demonstrate the ability to bypass not only KASLR but also heap cookie protector. Among these 27 vulnerabilities, 12 vulnerabilities also provide us with the ability to uncover stack canary and 8 vulnerabilities also exhibit the capability of performing arbitrary kernel read. These observations indicate that elastic kernel objects could generally make existing kernel protection futile.

Exploit diversity. From Table 2 & 7, 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

[6] FreeBSD has no heap cookie protection and thus no security impact on heap protector.

(e.g., the one indicated by CVE-2017-7184 listed in Table 7). From the column "Capability" in both tables, we can 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, it 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, it is because the richness potentially diversifies the way to craft a working exploit, making the pattern-based exploitation detection more challenging.

Analysis of failure cases. For the 13 vulnerabilities that ELOISE 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, CVE-2018-1000112, and 3d67[68], 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, CVE-2019-6225, and 422a[69], while their PoC programs demonstrate the ability to corrupt some data in general caches/zones, ELOISE cannot track down any kernel object with its length field overlapping with the corrupted region. For the vulnerability indicated by CVE-2018-4243, although ELOISE identifies objects overlapping with the corrupted region, the vulnerability provides only the ability to overwrite the length field with all zeros. Regarding vulnerabilities associated with CVE-2019-5603, CVE-2019-5596, and bf96[74], the corruption happens in special caches/zones in which no elastic objects are available for further exploitation.

5.4 Results of Experiment III

The A/B test experiment results show that Group A, equipped with ELOISE, succeeded in disclosing the base address of kernel image and bypassing KASLR for all 5 vulnerabilities, whereas Group B failed all. To get insights on this significant difference, we reviewed the surveys gathered from both groups while they analyzed vulnerabilities and developed exploits. For more detailed results, readers could refer to Appendix A.5.

From our collected survey results, we found that it took roughly 0.5~2 hours for the two groups to finish exploring the capability of one vulnerability. On the one hand, it indicates that the two groups have no apparent difference in expertise level. On the other hand, this result shows that the capability exploration is not a heavy workload for security analysts with Linux kernel debugging experience. This conclusion aligns with our post-test survey, in which all participants stated that the capability exploration is not a challenging task when the corresponding PoC program is provided. They reported that after the PoC program is given, they can rely on the built-in debugging features in the kernel (e.g., KASAN) to quickly learn which cache is corrupted and which part of the memory is corrupted. Then, they can disable KASAN, use GDB to trace kernel execution, and thus determine the value of overwritten data under their control.

From our collected survey (summarized in Table 9), we also discovered that under the guidance of our tool ELOISE, Group A could complete the identification of elastic objects in less than 0.5 hours. In contrast, Group B was stuck in this identification stage and made no progress for any of these 5 vulnerabilities in 24 hours. This significant difference implies the ELOISE's benefits in identifying elastic objects in the kernel and facilitating the exploitation. Following this observation, we also reviewed our post-test survey results. We observed that Group B thought the most challenging part of the exploitation is searching for the elastic objects and "felt frustrated" when facing the large codebase. For Group A, the post-test survey indicates that the most challenging part for exploit development is how to stabilize exploitation. They stated that unexpected kernel activities could intervene in the heap layout manipulation, making the disclosed data useless for bypassing KASLR. For example, Group A reported that their exploits leak all zero or other trash value from time to time. They put lots of effort into taming noisy kernel activities.

Based on our A/B test experiment, we argue that ELOISE is beneficial for exploit development. Although it is not an end-to-end automation tool, it could save security analysts' efforts to search for the elastic objects in the kernel and match them with the vulnerabilities. Using ELOISE, analysts could craft a working exploit more efficiently.

6 DEFENSE MECHANISM

In this section, we first discuss existing defense mechanisms. Then, we describe the design, implement and evaluation of our defense approach. Due to the space limit, we leave alternative defense mechanisms and future research in Appendix A.4.

6.1 Existing Defense Mechanisms

Recall that the exploitation method discussed in this paper requires manipulating the kernel heap layout. Intuition suggests that the existing defense most likely to mitigate this exploitation method is heap freelist randomization [16, 23]. However, this approach cannot be an effective solution to our problem. One is because research [16] has already demonstrated that freelist randomization has no effects on mitigating exploitation against use-after-free and double-free vulnerabilities. The other is because there have already been many techniques proposed for bypassing this mitigation effectively (e.g., [6, 28, 44]).

In addition to memory layout manipulation, the exploitation method also needs to accurately locate and modify the length field in an elastic object. As a result, another possible existing defense is structure layout randomization [15], which shuffles the fields in a data structure at the compilation phase for preventing attackers from predicting the offset of sensitive data within the structure. 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 [35] 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 [32].

Apart from above defenses, both Linux and XNU kernel import "USERCOPY" checking [22] from PaX/Grsecurity team. This hardening ensures that the length argument does not exceed the size of cache/zone slot or stack frame. While this technique can mitigate the threat of some elastic objects, it suffers from two problems. On the one hand, it only enforces the length checking for `copy_{from/ ↪ to}_user()` and `copyout()`. Other critical kernel functions for data transferring are not included. On the other hand, the legit length range is not restricted enough. It is still possible to leak sensitive data residing in the cache/zone slot or stack frame.

6.2 Our Defense Approach

Design. To mitigate the threat of elastic objects, we propose a new defense mechanism. It isolates elastic objects that ELOISE identifies into individual shadow caches/zones. To be specific, we create an isolated shadow cache/zone (e.g., `kmalloc-isolated-16`) for each general cache/zone (e.g., `kmalloc-16`). Using shadow caches/zones, we store elastic objects with the corresponding sizes. For example, the elastic object `ldt_struct` originally allocated in `kmalloc-16` will be assigned in `kmalloc-isolated-16` after the isolation mechanism is enabled.

With the isolation mechanism, an adversary has little chance to leverage the vulnerability tied to other objects to manipulate the length (and pointer) field in elastic objects. Besides, common heap spray objects and kernel objects with sensitive information like function pointers are also isolated from the elastic objects. They could not be used for heap Fengshui and spraying. Admittedly, an elastic object itself could also be potentially vulnerable, which provides attackers with the ability to overwrite other elastic objects sharing the same isolated shadow cache/zone. However, as showed in the following section, vulnerable elastic objects are relatively fewer than non-elastic vulnerable objects. Therefore, the cache-isolation-based defense dramatically raises the bar for launching the exploitation method and reduces leaked data's significance. Note that our defense approach is very different from a recently proposed isolation mechanism – xMP [58]. xMP provides page granularity isolation. With such isolation granularity, overwrite/overread still work, and the exploitation method is still effective.

Implementation. We implemented and prototyped the proposed isolation-based mitigation in the Linux kernel. To be specific, we added the support of our mitigation method by creating the shadow caches and other caches at the boot time. Further, we modified the kernel source code by adding one more flag (e.g., `__GFP_ISOLATE`). In our implementation, we used this flag as an additional argument for the functions that allocate kernel objects (e.g., `kmalloc()`). Using this flag as an indicator, our modified kernel could determine if the allocated objects should be placed in the shadow caches. To determine which allocation should happen at the shadow cache, we use the output of ELOISE. It indicates which calls indeed allocate elastic objects. In this way, we can ensure that the elastic objects can be isolated from other kernel objects physically.

Performance Evaluation. We evaluate the performance overhead of the proposed mitigation on a machine with a 1.6 GHz CPU, 16GB RAM, and 500 GB HDD. Our hardened kernel is modified from a plain Linux kernel, which is v5.5.3 (same as the kernel version used in our previous experiment). We conducted the measurements using three sets of benchmarks. The first set is micro-benchmarks from LMbench v3.0 [51], which tests the latency and bandwidth of common system calls and I/O operations. The second set is macro-benchmarks from Phoronix Test Suite 9.8 [1], which runs five real-world applications. To prevent the overhead from being hidden behind sophisticated kernel execution, we especially designed the third set benchmark to stress-test the impact of our mitigation approach. This set of customized benchmarks uses the system call sequences to reach elastic object allocation and corresponding data leakage intensively. In our experiment setup, we ran the three sets of benchmarks for three rounds and calculated the average. Due to the space limit, we list the detailed results in Table 8 (in the Appendix). Overall, we could observe that the performance overhead is negligible, with the average 0.19% performance drop. From Table 8, we could also find that for TCP socket I/O throughput, the hardened kernel even performs better (6.29% improvement). This fluctuation is presumably because our mitigation approach changes the hit rate of hardware cache, and the deviation of benchmarks adds uncertainty to the measurement.

Security Evaluation. We also evaluated the security of our proposed mitigation approach by re-pairing the vulnerabilities with elastic objects. For all Linux vulnerabilities shown in Table 7 (except for `CVE-2017-7184` and `CVE-2017-17053`), ELOISE no longer reports elastic objects available for performing the exploitation after our isolation mechanism is applied. It is because the elastic objects and vulnerable objects are mostly different. They are isolated into two caches. There is no longer a possibility to use vulnerable objects to manipulate the length field of an elastic object. For vulnerable objects in `CVE-2017-7184` and `CVE-2017-17053`, they are also elastic objects allocated in the shadow caches. Technically, they can be leveraged to overwrite data in the isolated caches and thus manipulate the length field of an elastic object for data disclosure. However, we argue that, even if this situation exists, it does not dilute our proposed defense method because the disclosed data is not likely to be useful for bypassing kernel mitigation. Taking the practice of circumventing KASLR using `CVE-2017-17053` as an example, to use the vulnerable object to reveal a kernel base address, in addition to leveraging the elastic object, an attacker usually has to identify a general object that encloses a function pointer. Then, the attacker needs to place the object in the same isolated cache. However, due to general and elastic kernel object isolation, such an object is no longer available for this isolated cache.

7 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. Side-channel based attack [24, 34] is one common approach for exploitability escalation. Technically, it leverages 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 [37]. Gruss *et al.* propose to utilize pre-fetch instructions to circumvent KASLR without triggering SMEP/SMAP protection [27]. Lipp *et al.* introduce a method to read arbitrary kernel memory from userland by exploiting out-of-order execution in modern processors [46].

Another exploitability escalation method is through new exploitation approaches. For example, ret2dir [39] injects exploitation payload to physmap instead of user space to circumvent SMEP and SMAP. To bypass a series of Linux kernel protection (except for KASLR), the technique KEPLER [79] 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 [9] exploits the vulnerable software through corrupting data flow instead of triggering critical control flow examination.

Facilitating exploit development. Researchers have proposed many exploitation automation techniques, ranging from the works that assemble exploits fully automatically (e.g., [4, 7, 8, 33, 62, 63, 66]) to the works that partially facilitate exploit development (e.g., [5, 29, 30, 36, 59, 78, 82]). However, 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 [81] 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 [49]. Following this, Cho *et al.* further extend the stack spraying method in [14]. To expedite the exploration of useful primitives for kernel Use-After-free exploitation, FUZE [80] 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 [10]. To obtain the desired heap layout for kernel exploitation, SLAKE [11] proposes a method to navigate kernel objects and then an algorithm to elastic kernel layout automatically.

Uniqueness of our work. 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.

8 CONCLUSION

Using elastic objects to bypass kernel protection is a commonly adopted exploitation practice. However, no systematic research has been conducted to study the effectiveness of this exploitation method. As such, it has not yet raised sufficient awareness and motivates the development of a defense against such exploitation. In this work, we show that elastic objects could nearly always facilitate a kernel vulnerability to bypass exploitation mitigation such as KASLR, heap cookie protector, and stack canary. Taking a close look at existing kernel defense mechanisms, we discover that none can be useful or practical for hindering the threat of elastic kernel objects. Inspired by this finding, we introduce a new lightweight defense mechanism. We conclude that the threat of elastic objects can be, to some extent, mitigated if the kernel could place elastic objects into separated caches or zones.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedback. This work was supported by the 2020 IBM PhD Fellowship Program, NSF 1718459, and ONR N00014-20-1-2008.

REFERENCES

[1] 2015. Phoronix Test Suite. http://www.phoronix-test-suite.com/.
[2] 2019. Code and Exploits for ELOISE. https://github.com/chenyueqi/w2l.
[3] 0x3f97. 2018. cve-2017-8890 root case analysis. https://0x3f97.github.io/exploit/2018/08/13/cve-2017-8890-root-case-analysis/.
[4] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS).
[5] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David 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).
[6] Ian Beer. 2017. Exception-oriented exploitation on iOS. https://googleprojectzero.blogspot.com/2017/04/exception-oriented-exploitation-on-ios.html.
[7] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang 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).
[8] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P).
[9] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In Proceedings of the 14th USENIX Security Symposium (USENIX Security).
[10] Weiteng Chen, Xiaochen Zou, Guoren Li, , and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In Proceedings of the 29th USENIX Security Symposium (USENIX Security).
[11] Yueqi Chen and Xinyu 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).
[12] Yueqi Chen, Xinyu Xing, and Jimmy 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.
[13] Eric Chiang. 2019. User Namespaces. https://ericchiang.github.io/post/user-namespaces/.
[14] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2020. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. In 14th USENIX Workshop on Offensive Technologies (WOOT).
[15] Kees Cook. 2017. security things in Linux v4.13. https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/.
[16] Kees Cook. 2017. security things in Linux v4.14. https://outflux.net/blog/archives/2017/11/14/security-things-in-linux-v4-14/.
[17] Jonathan Corbet. 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/.
[18] Jonathan Corbet. 2016. Exclusive page-frame ownership. https://lwn.net/Articles/700647/.
[19] Jonathan Corbet. 2017. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/.
[20] SSD Secure Disclosure. 2017. SSD Advisory – Linux Kernel AF_PACKET Use-After-Free. https://ssd-disclosure.com/archives/3484.
[21] dp304. 2018. Alternative to flexible array members for avoiding multiple allocations. https://www.gamedev.net/forums/topic/696730-alternative-to-flexible-array-members-for-avoiding-multiple-allocations/.

[22] Jake Edge. 2016. Hardened usercopy. https://lwn.net/Articles/695991/.
[23] Stefan Esser. 2016. iOS 10 - Kernel Heap Revisited.
[24] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael 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).
[25] Google. 2019. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.
[26] Google. 2020. syzbot Dashboard. https://syzkaller.appspot.com/upstream.
[27] Daniel Gruss, Clémentine Maurice, and Anders 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).
[28] Mathieu Hautebas. 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.
[29] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic Heap Layout Manipulation for Exploitation. In Proceedings of the 27th USENIX Security Symposium (USENIX Security).
[30] Sean Heelan, Tom Melham, and Daniel Kroening. 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).
[31] Jann Horn. 2018. A cache invalidation bug in Linux memory management. https://googleprojectzero.blogspot.com/2018/09/a-cache-invalidation-bug-in-linux.html.
[32] Jann Horn. 2020. Linux Email list: CONFIG_DEBUG_INFO_BTF and CONFIG_GCC_PLUGIN_RANDSTRUCT. https://www.spinics.net/lists/bpf/msg16648.html.
[33] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-oriented Exploits. In Proceedings of the 24th USENIX Security Symposium (USENIX Security).
[34] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P).
[35] Nur Hussein. 2017. Randomizing structure layout. https://lwn.net/Articles/722293/.
[36] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS).
[37] Yeongjin Jang, Sangho Lee, and Taesoo 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).
[38] M. Jones. 2010. User space memory access from the Linux kernel. https://developer.ibm.com/technologies/linux/articles/l-kernel-memory-access/.
[39] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security).
[40] Kengiter and adityapakki. 2019. Crix: Detecting Missing-Check Bugs in OS Kernels. https://github.com/umnsec/crix.
[41] Andrey Konovalov. 2017. Exploiting the Linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.
[42] Andrey Konovalov. 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.
[43] Andrey Konovalov. 2018. A proof-of-concept exploit for CVE-2017-18344. https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-18344/poc.c.
[44] Azeria Labs. 2020. Grooming the iOS Kernel Heap. https://azeria-labs.com/grooming-the-ios-kernel-heap/.
[45] Lexfo. 2018. CVE-2017-11176: A step-by-step Linux Kernel exploitation. https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html.
[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium (USENIX Security).
[47] Kangjie Lu and Hong 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).
[48] Kangjie Lu, Aditya Pakki, and Qiushi 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).
[49] Kangjie Lu, Marie-Therese Walter, David Pfaff, and Stefan 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).
[50] Wolfgan Mauerer. 2008. Professional Linux Kernel Architectures. Chapter 12.11.
[51] Larry McVoy and Carl Staelin. 2015. LMbench - Toos for Performance Analysis. http://lmbench.sourceforge.net/.
[52] Patrick Mochel and Mike Murphy. 2020. sysfs - The filesystem for exporting kernel objects. https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.
[53] Vitaly Nikolenko. 2016. CVE-2016-6187: Exploiting Linux kernel heap off-by-one. https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit.
[54] Vitaly Nikolenko. 2018. Dissecting a 17-year-old kernel bug. https://duasynt.com/slides/bevx-talk.pdf.
[55] Vitaly Nikolenko. 2018. Linux Kernel universal heap spray. https://duasynt.com/blog/linux-kernel-heap-spray.
[56] OpenWall. 2020. John the Ripper password cracker. https://www.openwall.com/john/.
[57] Alexander Popov. 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.
[58] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: selective memory protection for kernel and user space. In Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P).
[59] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. 2017. Modular Synthesis of Heap Exploits. In ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS).
[60] Microsoft Research. 2020. Z3. https://github.com/Z3Prover/z3.
[61] Chris Salls. 2017. Exploiting CVE-2017-5123 with full protections. SMEP, SMAP, and the Chrome Sandbox! https://salls.github.io/Linux-Kernel-CVE-2017-5123/.
[62] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS).
[63] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni 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).
[64] Spudd86. 2010. Flexible array member in C-structure. https://stackoverflow.com/questions/3047530/flexible-array-member-in-c-structure.
[65] Richard M. Stallman. 2019. GNU Debugger. https://www.gnu.org/software/gdb/.
[66] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS).
[67] syzbot. 2018. KASAN: slab-out-of-bounds Write in crypto_dh_encode_key. https://syzkaller.appspot.com/bug?id=a84d6ad70b281bfc5632f272f745104fb43d219d.
[68] syzbot. 2018. KASAN: slab-out-of-bounds Write in mpol_parse_str. https://syzkaller.appspot.com/bug?id=3d67d693e0529df8ac89ba55b00b54e5d967e021.
[69] syzbot. 2018. KASAN: slab-out-of-bounds Write in pipe_write. https://syzkaller.appspot.com/bug?id=422a020e119fbac4c15d8fed114cc1696fe5c51a.
[70] syzbot. 2018. KASAN: slab-out-of-bounds Write in sha512_final. https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c.
[71] syzbot. 2018. KASAN: use-after-free Read in __lock_acquire (2). https://syzkaller.appspot.com/bug?id=1379b6b21a2ffecd1ea4e2b564cc7e35d9f388b2.
[72] syzbot. 2018. KASAN: use-after-free Read in snd_timer_open. https://syzkaller.appspot.com/bug?id=e9287fe57ad2f862eedb05012481132486f3b887.
[73] syzbot. 2018. KASAN: use-after-free Write in bpf_tcp_close. https://syzkaller.appspot.com/bug?id=6a6fd266a962be281b17c864a073675150e36ca5.
[74] syzbot. 2018. KASAN: use-after-free Write in dst_release. https://syzkaller.appspot.com/bug?id=bf967d2c5ba62946c61152534c8b84823d848f05.
[75] syzbot. 2019. KASAN: use-after-free Write in __xfrm_policy_unlink. https://syzkaller.appspot.com/bug?id=ebeba334a8a886e3d5dc25641e201e894d4d9657.
[76] syzbot. 2020. KASAN: use-after-free Read in route4_get. https://syzkaller.appspot.com/bug?id=5bb09c0c5b65ab2ce628ba26fe7cbd06144bd952.
[77] PaX Team. 2000. Design & implementation of PAGEEXEC. .
[78] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, BingChang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In Proceedings of the 25nd ACM SIGSAC Conference on Computer and Communications Security (CCS).
[79] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In Proceedings of the 28th USENIX Security Symposium (USENIX Security).
[80] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Wei Zou, and Xiaorui Gong. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In Proceedings of the 27th USENIX Security Symposium (USENIX Security).
[81] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu 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).
[82] Insu Yun, Dhaval Kapil, and Taesoo Kim. 2020. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In Proceedings of the 29th USENIX Security Symposium (USENIX Security).

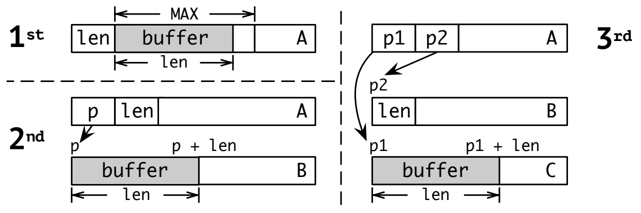


Figure 5: The alternative implementations of elastic kernel objects.

A APPENDIX

A.1 Implementation of Elastic Kernel Objects

The kernel object `xfrm_replay_state_esn` shown in Figure 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 Linux, FreeBSD, and XNU, we also discover three alternative implementations. Here, we summarize them below.

As is illustrated in the first example shown in Figure 5, 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.

As is depicted in the second and third examples shown in Figure 5, 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` [52] 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, the exploitation method takes advantage of these unprivileged channels to uncover kernel data to userland processes. In the following, we summarize these communication channels and categorize them into three types.

User Space Memory Access APIs [38]. User space memory access APIs are those functions like `copy_to_user()` and `copyout()`. 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, the exploitation method cannot manipulate such APIs. In this work, we exclude such non-manipulable APIs and list only those manipulable ones in Table 3.

Netlink socket family. This channel uses the networking framework for kernel-user communication. Designed as a generic object model [50], `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 Section 5, 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. In Table 3, 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 3, we list all the functions in this type and specify the number of data that the kernel can capsule into network packets.

[7]FreeBSD has no heap cookie protection and thus no impact on heap protector.

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

Linux					
Cache	**Struct**	**Offset (len/ptr)**	**Potential**	**Privilege**	**Constraints**
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 anchor2: S	∅	[8, 9) < Arg, anchor1 in put_cmsg() [8, 9) ≠ 0, anchor2 in do_ip_getsockopt()
	iovec	[8, 16)/[0, 8)	H & A	∅	∅
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, 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, [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
kmalloc-1024	xfrm_policy	[372, 373)/NA	S	NET_ADMIN	∅
	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 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, [96, 104) == kaddr

Table 4: Elastic kernel objects identified and confirmed in Linux. 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 Section 5.

A.3 Detail of Evaluation

Elastic object identification. Table 4~6 list 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

XNU					
Cache	**Struct**	**Offset (len/ptr)**	**Potential**	**Privilege**	**Constraints**
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 5: Elastic kernel objects identified and confirmed in XNU. 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 Section 5.

FreeBSD					
Cache	**Struct**	**Offset (len/ptr)**	**Potential**	**Privilege**	**Constraints**
kmem.16	TWE_Param★†	[3, 4)/NA	H	∅	p ≠ null, [3,4) ≤ twe_paramcommand.3
	iovec	[8, 16)/[0, 8)	H & A	∅	∅
	sockaddr	[0, 1)/NA	H	∅	∅
kmem.32	i40e_nvm_access	[12, 16)/NA	H	∅	[12, 16) > 0, [12, 16) < 4097
	vpd_readonly	[16, 20)/[8,16)	H & A	∅	∅
	vpd_writeonly	[20, 24)/[8,16)	H & A	∅	∅
kmem.64	uio	[24, 32)/NA	H	∅	[32, 36) ≠ 2
	gctl_req_arg	[28, 32)/[40, 48)	H & A	∅	[24, 28) == 32
	ips_ioctl	[20, 24)/[8, 16)	H & A	∅	∅
kmem.128	usb_symlink	[80, 82)/NA	H	∅	p ≠ null, [80, 81)+[81, 82) ≤ 252
kmem.256	ucred	[52, 56)/[176, 184)	H & A	∅	∅
	shmfd	[0, 8)/NA	H	∅	∅
	iso_node†	[56, 64)/NA	H	∅	[56, 64) ≤ uio.2
	iso_mnt†	[48, 52)/NA	H	∅	[48, 52) ≤ uio.3
kmem.512	acc_fib†	[8, 10)/NA	H	∅	[8, 10)≤ aac_softc.61
	dirent	[20, 22)/NA	H	∅	[20, 22) ≤ Arg
kmem.1024	buf	[48, 52)/[24, 32)	H & A	∅	∅
kmem.2048	fw_device	[16, 20)/NA	H	∅	p ≠ null,[16, 20) ≥ 1024
mbuf	mbuf	[24, 28)/[8, 16)	H & A	∅	∅
TMPFS node	tmpfs_node	[40, 48)/NA	H	∅	∅

Table 6: Elastic kernel objects identified and confirmed in FreeBSD. 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 Section 5.

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 4~6, for clarification, we mark all the special caches/-zones with a triangle symbol △ and highlight with a star symbol ★

CVE-ID or Syzkaller-ID	Type	Capability	Suitable objects #	Security Impact
FreeBSD				
2019-5603	UAF	file_zone:[40, 44)=*	0	NA
2019-5596	UAF	file_zone:[40, 44)=*	0	NA
2016-1887	OOB	zone_mbuf:[0, 256)=*	1	BA & AR [7]
Linux				
1379... [71]	OOB	kmalloc-512:[0, 512)=*	10 + (1)	SC, HC, BA
3d67... [68]	OOB	NA	0	NA
422a... [69]	OOB	kmalloc-64:[0, 4)=0x8	0	NA
5bb0... [76]	UAF	kmalloc-192:[16, 24)=0, kmalloc-192:[48, 56)=kaddr	1	HC, BA
6a6f... [73]	UAF	kmalloc-1024:[0, 8)=kaddr	3	HC, BA
a84d... [67]	OOB	kmalloc-32:[0, 4)=*	1	HC, BA
bf96... [74]	UAF	ip_dst_cache:[64, 68)=*	0	NA
e4be... [70]	OOB	kmalloc-64:[0, 16)=*, [16, 24)=192, [24, 64)=0	6	SC, HC, BA
e928... [72]	UAF	kmalloc-256:[120, 128)=kaddr	1	HC, BA, AR
ebeb... [75]	UAF	kmalloc-1024:[15, 24)=kaddr	1	HC, BA
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)=*	4	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)=*	24 + (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 7: 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 Section 5.

the objects that could belong to all the caches/zones greater than they are specified in the table. Besides, we utilize the symbol ∅ to signify no privilege is required if an attacker performs exploitation with the corresponding object. Similarly, the same symbol ∅ 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≠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 [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.

Kernel vulnerabilities and their exploitability. Table 7 lists all the kernel vulnerabilities used for our evaluation. From the column on the left to the right, the results shown in the table indicate (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 7, as is specified in Section 3.3, 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 Section 4, when pairing a vulnerability with elastic objects, ELOISE ignores the paths involving variables and discard corresponding kernel objects (if for that elastic object ELOISE 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 7, even without using objects in this type, vulnerabilities can still find alternative objects for performing successful exploitation.

As is shown in Table 7, the vulnerabilities selected for our evaluation 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).

Vulnerability capability summarization. Table 7 also summarizes the capability of each vulnerability. In our evaluation, 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 [53–55]). We will make all these vulnerabilities available in virtual machines and release the exploits crafted by using elastic kernel objects.

Performance overhead of our proposed defense. Table 8 lists the performance of the Linux kernel with and without our proposed defense mechanism. The results are observed from three different benchmarks discussed in Section 6.2. We present the performance change in the column of "overhead". For latency measure, a negative percentage indicates performance improvement, whereas the positive percentage represents the performance degradation. For the throughput measure, the negative and positive rates mean performance degradation and increase, respectively.

A.4 Alternative Defense Methods

In addition to the defense proposed in Section 6, there are alternative defense solutions that might be useful for mitigating the threat of elastic kernel objects. In the following, we discuss these methods and analyze the challenges of their implementation.

The first possible solution is to 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 in an elastic buffer 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 memory and performance overhead, the key challenge of this solution is to develop a lightweight method to minimize overhead in a systematic method.

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 elastic structures and introduce a checksum field. Then, when the kernel allocates the corresponding object and initializes its length field, we could encrypt the length value and store it in the checksum field accordingly. With this design, at the time of disclosing data in the elastic buffer to the userland, the kernel could easily retrieve and scrutinize the checksum. However, the key challenge of implementing this idea is to ensure the addition of the checksum will not influence the usability of the kernel. For example, some elastic data structures designed for protocols have specific formats. After allocating objects in these types, the kernel references the data through corresponding offsets. If introducing additional field into such objects, one has to ensure the newly added checksum field does not incur incorrect data reference.

A.5 More Details of User Study

Section 5.1 describes the setup of our user study. Here, we provide the three survey forms (i.e., Figure 6, 7, and 8) we used during the experiment. In Table 9, we display the time took for each group spent in solving the five vulnerabilities. As the short probing survey

1. Did you ever debug Linux kernel vulnerabilities before?
   a. Yes
   b. No
2. How long is your experience in Linux kernel security?
3. Did you ever write an exploit for Linux kernel vulnerability?
   a. Yes
   b. No
4. If you answer yes to 3, how do you usually bypass KASLR
   a. I assume no KASLR
   b. I use hardware side-channel
   c. I use information disclosure in `dmesg`
   d. I bypass KASLR if the vulnerability provides read primitive
   e. Others
5. Do you know or ever use elastic structures for KASLR bypassing?
   a. Yes
   b. No
6. Please list the CVE-ID or other ID or links of vulnerabilities you have debugged or drafted exploits for.

Figure 6: Self-assessment form.

1. Which vulnerability are your group working on?
2. For this vulnerability, which stage are your group on?
   a. Vulnerability capability exploration
   b. Elastic object identification
   c. Memory layout manipulation.

Figure 7: Short probing survey form.

1. Do you think exploring the capability of vulnerability is difficult?
   a. Yes
   b. No
2. What methods do you usually use to explore the capability of vulnerability?
3. What do you think is the most challenging part of drafting the exploits?

Figure 8: Post-test survey form.

	2010-2959		2017-7184		2017-7308		2017-8890		ebeb...[75]	
Stage	A	B	A	B	A	B	A	B	A	B
1	1	0.5	2	2	2	2	1	1	1	1
2	0	4.5	0	2	0	3.5	0	4.5	0	3
3	1	NA	2	NA	2	NA	1	NA	1	NA

Table 9: Summary of our probing survey results. The # in the table indicates how many hours reported through the short probing survey that the two groups spent on the stage of a specific vulnerability. NA means the corresponding group participant did not enter the stage while they develop the exploit for a particular vulnerability.

Benchmark	w/o defense	w/ defense	Overhead
LMbench - latency (ms)			
syscall()	0.3813	0.3796	-0.46%
open()/close()	1.5282	1.5290	0.05%
read()	0.4596	0.4529	-0.94%
write()	0.4125	0.4127	0.05%
select() (10 fds)	0.5114	0.5043	-1.39%
select() (100 fds)	1.1805	1.1774	-0.26%
stat()	0.7590	0.7600	0.14%
fstat()	0.4576	0.4584	0.19%
fork() + exit()	90.37	91.71	1.46%
fork() + execve()	255.18	257.85	1.05%
fork() + /bin/sh	858.86	863.77	0.57%
sigaction()	0.4182	0.4192	0.25%
Signal delivery	0.9337	0.9309	-0.30%
Protection fault	0.6914	0.7093	2.58%
Pipe I/O	3.7497	3.7951	1.87%
UNIX socket I/O	5.9786	5.882	-1.62%
TCP socket I/O	9.7846	9.6776	-1.09%
UDP socket I/O	6.5358	6.2251	-4.75%
LMbench - throughput (MB/s)			
Pipe I/O	4755.49	4753.89	0.03%
UNIX socket I/O	10385.07	10307.40	0.75%
TCP socket I/O	6327.32	6725.17	-6.29%
mmap() I/O	13559.20	13511.95	0.35%
File I/O	7707.81	7702.82	0.06%
Phoronix - latency (s)			
FFmpeg	14.01	14.46	3.22%
GnuPG	17.39	17.35	-0.22%
Phoronix - throughput			
Apache (request/s)	16700.23	16088.00	3.67%
OpenSSL (signs/s)	272.00	272.00	0
7-Zip (MIPS)	9970.00	9374.00	5.98%
Customized bench - latency (ms)			
sock_fprog_kern	28.54	28.30	0.09%
ldt_struct	33.81	31.48	-2.52%
ip_options	29.29	30.67	2.40%
user_key_payload	34.04	35.33	-2.87%
xfrm_replay_state_esn	29.69	30.06	1.67%
ip_sf_socklist	29.13	28.05	-3.78%
sg_header	31.84	30.75	-2.99%
inotify_event_info	32.68	31.77	0.42%
msg_msg	27.75	26.83	0.66%
tcp_fastopen_context	28.79	28.65	-1.04%
request_key_auth	81.23	79.88	2.98%
xfrm_algo_auth	30.32	29.50	-0.28%
xfrm_algo	28.64	28.43	-0.11%
xfrm_algo_aead	31.36	31.39	0.13%
xfrm_policy	31.07	30.53	-1.43%
Average			0.19%

Table 8: The performance of the Linux with and without our proposed defense. For latency measure, the smaller the number is, the better the performance is. For throughput, the larger the number is, the better the performance is.

is collected every 30 minutes, the time is accumulated according to the survey results.