AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects

Ruipeng Wang, *National University of Defense Technology;* Kaixiang Chen and Chao Zhang, *Tsinghua University;* Zulie Pan and Qianyu Li, *National University of Defense Technology;* Siliang Qin, *University of Chinese Academy of Sciences;* Shenglin Xu, Min Zhang, and Yang Li, *National University of Defense Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/wang-ruipeng

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects

Ruipeng Wang[1†] Kaixiang Chen[2] Chao Zhang[2] Zulie Pan[1] Qianyu Li[1]
Siliang Qin[3] Shenglin Xu[1] Min Zhang[1*] Yang Li[1]
[1]*National University of Defense Technology* [2]*Tsinghua University*
[3]*University of Chinese Academy of Sciences*
{wangruipeng, panzulie17, liqianyu, xushenglin, zhangmindy, liyanghf}@nudt.edu.cn>
ckx18@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn
qinsiliang18@mails.ucas.ac.cn

Abstract

Memory corruption vulnerabilities are often exploited to corrupt sensitive objects and launch attacks. An efficient way to mitigate such threats is identifying and protecting such sensitive objects against corruption. However, it is still an open question that what objects are security sensitive and how sensitive they are. In this paper, we present the first expert system based solution AlphaEXP to identify security sensitive objects, in a specific and important target – the Linux kernel. It works by simulating an adversary to assess whether an object could be abused to get unintended capabilities and contribute to exploitation, and marks it as sensitive if so. Specifically, AlphaEXP first constructs a knowledge graph to represent the facts of the kernel, including objects, functions, and their relationships etc. Then, it explores the knowledge graph to infer potential attack paths for given vulnerabilities, and marks objects used in the attack paths as sensitive. Lastly, it evaluates the feasibility of the attack paths in a customized emulating system, and classifies the sensitivity of objects accordingly. We have built a prototype of AlphaEXP and evaluated it on 84 synthesized *representative* vulnerabilities and 19 real world vulnerabilities to identify sensitive kernel objects. AlphaEXP successfully generates attack paths for most of these vulnerabilities, and finds 50 objects that could be abused to get writing capability, 81 objects with reading capability, and 112 objects with execution capability, then classifies them into 12 levels of sensitivity.

1 Introduction

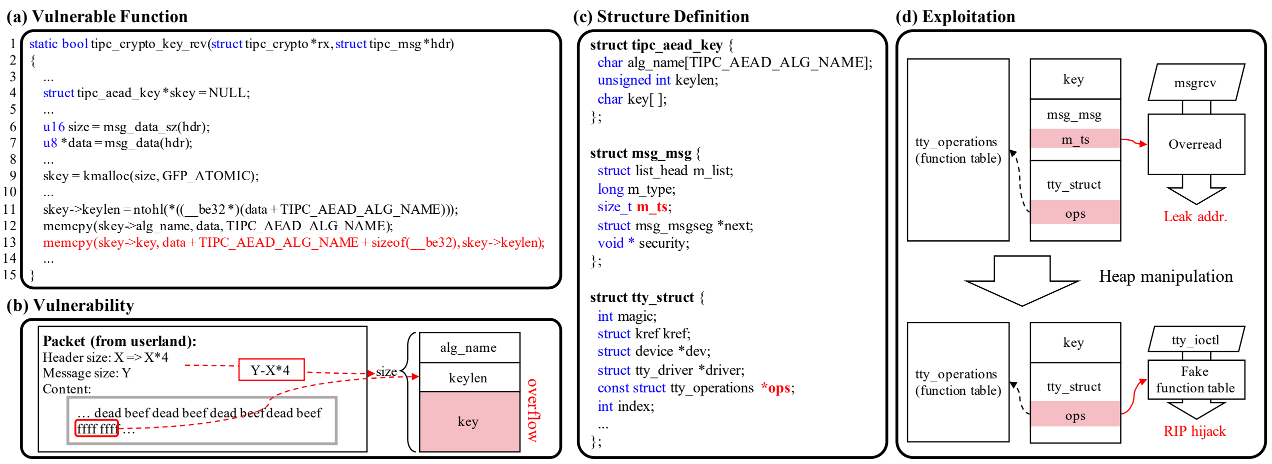
Memory corruption vulnerabilities are one of the major threats to software. Adversaries often exploit such vulnerabilities to corrupt sensitive objects and launch attacks, including information leakage, privilege escalation, and control flow hijacking. To mitigate threats of memory corruption vulnerabilities, there are three types of solutions proposed and deployed in practice: vulnerability patching, software and system hardening, and object-specific protections.

Patching is the most straightforward and cost-effective solution to mitigate vulnerabilities. However, it cannot mitigate unknown 0-day vulnerabilities, and requires huge engineering efforts due to rapidly growing number of programs and vulnerabilities. On the other hand, software and system hardening is a promising solution to mitigate vulnerabilities including unknown ones. Such solutions in general check security invariants (*e.g.,* control flow integrity [1]) or set roadblocks (*e.g.,* DEP [2], ASLR [35]) along the attack paths of vulnerability exploitation and are not specific to vulnerabilities. However, such solutions would introduce performance costs to the system no matter the system is vulnerable or not.

Another solution that has a good balance between security and performance is object-specific protection. It works by hardening a limited number of sensitive objects (*e.g.,* by isolating them on a separate stack, CPI [29]) or checking their integrity (*e.g.,* by signing pointers with ARM PA [45]) to stop adversaries to corrupt them to launch attacks. For example, the iOS system utilizes ARM PA to protect the integrity of certain pointers abused by known exploits. However, the list of objects to protect [6] has to grow with the number of exposed exploits. To name a few, iOS 14.2 starts to protect `pipes` data pointers after the exploit [51], and iOS 14.5 signs ISA pointers after the iMessage exploit [20] is exposed.

Object-specific protection can effectively mitigate a set of exploits with low overheads, and thus is attractive to vendors. However, identifying sensitive objects to protect is an open question. There are three types of solutions: (1) Analyzing publicly exposed exploits to find out data that are abused. However, this solution heavily relies on the human experience, and cannot find sensitive data that have not been abused yet. (2) Classifying objects based on developers' intentions and the program's semantics. However, its results (*i.e.,* sensitive objects) may deviate from the adversary's. (3) Analyzing the target code following specific attack knowledge. For example, ELOISE [12] relies on information leak knowledge to identify elastic objects in kernel to bypass KASLR [17].

*Corresponding authors.
†Part of the work was done during the visit at Tsinghua University.



```
1  static bool tipc_crypto_key_rcv(struct tipc_crypto *rx, struct tipc_msg *hdr)
2  {
3      ...
4      struct tipc_aead_key *skey = NULL;
5      ...
6      u16 size = msg_data_sz(hdr);
7      u8 *data = msg_data(hdr);
8      ...
9      skey = kmalloc(size, GFP_ATOMIC);
10     ...
11     skey->keylen = ntohl(*((__be32 *)(data + TIPC_AEAD_ALG_NAME)));
12     memcpy(skey->alg_name, data, TIPC_AEAD_ALG_NAME);
13     memcpy(skey->key, data + TIPC_AEAD_ALG_NAME + sizeof(__be32), skey->keylen);
14     ...
15 }
```

```
struct tipc_aead_key {
  char alg_name[TIPC_AEAD_ALG_NAME];
  unsigned int keylen;
  char key[ ];
};

struct msg_msg {
  struct list_head m_list;
  long m_type;
  size_t m_ts;
  struct msg_msgseg *next;
  void * security;
};

struct tty_struct {
  int magic;
  struct kref kref;
  struct device *dev;
  struct tty_driver *driver;
  const struct tty_operations *ops;
  int index;
  ...
};
```

Figure 1: An example vulnerability from CVE-2021-43267 [34]. (a) shows the source code of the vulnerability. (b) shows how userland data triggers the vulnerability (d) shows how the exploit works [21], and (c) shows the definition of related objects.

SLAKE [13] follows heap manipulation knowledge to identify victim objects and spray objects. However, they are not generic solutions for identifying sensitive data, and cannot distinguish the sensitivity of the data.

Therefore, a general solution to identifying and classifying sensitive objects is highly demanded. To build such a solution, we have to answer two core questions: what are sensitive objects and how sensitive are they. First, there are thousands of objects in the kernel, and protecting all of them would have overwhelming overheads. Therefore, it is reasonable to only protect sensitive objects, *i.e., which could be abused by adversaries to get unintended capabilities and exploit a vulnerability*. Second, objects should be classified into different sensitivity levels, so that defenders could prioritize objects to protect when given limited resource budgets. We argue that objects are more sensitive *if attack paths that abuse them to launch attacks have a higher feasibility and impact.*

In this paper, we present the first expert system [31] based solution AlphaEXP to identify sensitive kernel objects. Given the definition of sensitive objects and sensitivity, AlphaEXP works by simulating an adversary to assess whether an object could be abused and the feasibility of abusing it to launch attacks. Informally, an adversary first analyzes the kernel and understands its objects and code, then explores the vulnerability's effect on these objects and hypothetically infers their further effects, and lastly finds a candidate attack path able to finish an exploitation. To simulate an adversary, AlphaEXP first constructs a knowledge graph to represent the facts of the kernel code, including objects, functions and their relationships etc. Then, it explores the knowledge graph from the starting point specified by the given vulnerability to infer potential attack paths[1]. Objects that are abused in the attack paths will be marked as sensitive. Further, AlphaEXP evaluates the feasibility of attack paths in a customized environment, and classifies the sensitivity of objects accordingly.

[1]AlphaEXP does not aim at generating exploits, and is not able to do so. Many engineering efforts are needed to construct exploits from attack paths. Thus, we ensure our work cannot be abused by adversaries that aim at generating exploits and mitigate the potential ethic issues.

We have implemented a prototype of AlphaEXP based on KINT [48], Syzkaller [19], and Soufflé [28], and evaluated it on 84 synthesized representative vulnerabilities and 19 real world CVE vulnerabilities. The results showed that, AlphaEXP could generate working attack paths for most of them, and finds 50 sensitive objects that could be abused to get writing capability, 81 with reading capability, and 112 with execution capability. Further, AlphaEXP classifies these sensitive objects into 12 sensitivity levels.

In summary, this paper makes the following contributions.

- We propose the first expert system AlphaEXP to identify sensitive kernel objects and classify their sensitivity, able to help defenders build cost-effective defenses.
- We construct a knowledge graph of the kernel with 358,383 entities and 440,741 relationships.
- We have implemented a prototype of AlphaEXP, and reported several hundreds of sensitive kernel objects and classified them into 12 sensitivity levels.

2 The Problem Statement

2.1 Running Example

As shown in Figure 1 (a), the vulnerability locates at line 13, and is triggered when when the value of user-controlled `skey->keylen` (line 11) exceeds the size of the buffer `skey->key`. Note that, as an elastic object, the size of the buffer is determined by the user, as shown in line 9 and Figure 1 (b). Moreover, the content to copy at line 13 is fully taken from userland. So, this vulnerability allows an adversary to write arbitrary values to out-of-bound memory.

The exploit shown in Figure 1 (d) makes use of structures `msg_msg` and `tty_struct` as victim objects. With elaborate

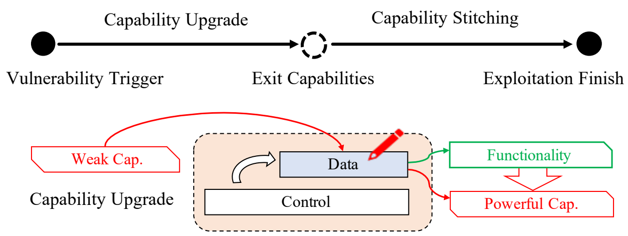


Figure 2: The sketchy procedure of exploitation.

heap manipulation, an object of msg_msg could be allocated following the overflowed object of tipc_aead_key. Then the field m_ts, which controls the read size of the msgrcv(), can be tampered by the vulnerability, which will cause an over-reading to leak information including the address of code and heap segment in tty_struct, to bypass the KASLR [17]. Similar to this process, an object of tty_struct could be allocated following tipc_aead_key, so that the vulnerability could tamper with the field ops, which is a vtable pointer. If any function in the vtable, *e.g.,* tty_ioctl(), is called, the program counter (PC) will be hijacked to execute ROP [36] gadgets after stack pivoting.

2.2 Challenges: Identifying Sensitive Objects

As shown in the running example, objects of the class msg_msg and tty_struct could be abused in an exploit to launch attacks, so they should be marked as security sensitive. But exploits are valuable and rare resources, we cannot rely on exposed exploits to identify all sensitive objects that would be abused by an exploit (maybe in the future).

Instead, we need a generic solution to identifying and classifying potential sensitive objects before they are abused by adversaries in the future. To this end, we have to answer two core questions: what are sensitive objects and how sensitive are they. It is not simple due to the following challenges:

First, heuristics-based solutions may have high false positives in recognizing sensitive objects. For instance, we cannot simply mark objects with a specific characteristic as sensitive, *e.g.,* whether they have pointer fields. There are thousands of types of objects in the kernel, which have all kinds of characteristics. For example, the structure of scsi_request is similar to msg_msg, and the operation blk_complete_sghdr_rq() has similar functionality as msgrcv. However, unlike msg_msg, object of scsi_request are not security sensitive, because we cannot find a time window to abuse scsi_request object.

Second, there are no quantitative methods to measure the sensitivity of objects yet. For example, msg_msg is widely used in exposed exploits [21,52]. Therefore, it should have a high sensitivity. But there are no metrics explaining why it is more sensitive and should be prioritized to protect.

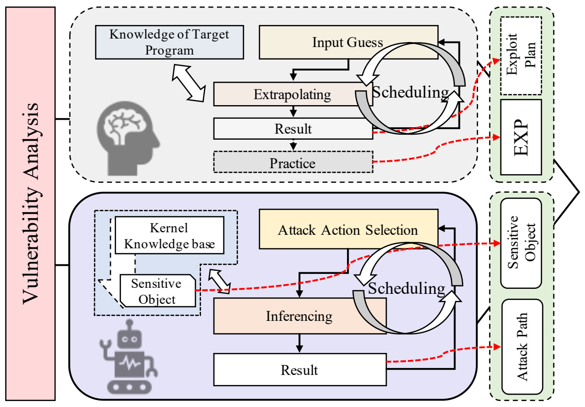


Figure 3: Illustration of the exploit generation procedure.

2.3 Characterizing Sensitive Objects

To identify sensitive objects, we have thoroughly analyzed the general procedure of kernel exploitation and how sensitive objects contribute to the procedure, to understand the characteristics of sensitive objects. As shown in Figure 2, the general exploitation procedure has two phases, *Capability Upgrade (CU)* and *Capability Stitching (CS)*.

After a vulnerability is triggered, the victim program enters a weird machine [16], where adversaries could get unintended capabilities, which are the capabilities of memory manipulation (*e.g.,* reading, writing and even executing memory) that are not intended by the program. In general, the entry capabilities that the adversary gets at the time of vulnerability triggering, which represents the consequence of abusing such objects, in general are *weak*, *i.e.,* the consequence is superficial. To launch attacks, the adversary first needs to enter the CU phase to upgrade entry capabilities to more powerful capabilities, and eventually to exit capabilities that are powerful enough to fulfill the requirements for exploitation. Specifically, there are two typical exit capabilities in kernel exploitation: ① the combination of reading capability and arbitrary code execution (ACE) capability; ① the combination of reading capability and arbitrary address writing (AAW) capability. They could be utilized to bypass mitigations like KASLR [17] and accomplish attacks like local privilege escalation.

As shown in Figure 2, unintended capabilities are upgraded gradually by the CU phase. In general, it utilizes the weak capability to tamper with certain data/objects, which will be accessed by legitimate code in the original program. The latter access will turn into a new functionality that the program does not intend to provide, and the adversary gets new and maybe stronger capabilities. *Objects that are tampered in the CU phase are therefore sensitive.*

After the entry capability is gradually upgraded to exit capabilities, the CS stage will try to stitch these capabilities together to construct a complete exploit, which could implant shellcode and achieve privilege escalation, etc.

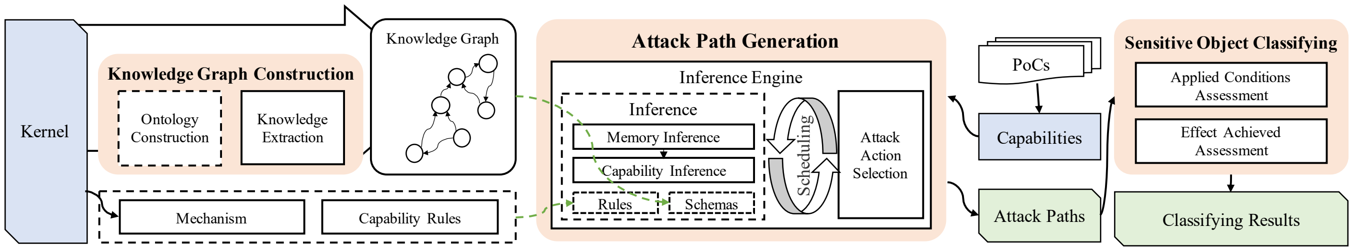


Figure 4: Overview of our solution AlphaEXP.

2.4 Intuition: Attack Simulation

Since sensitive objects are ones that could contribute to exploitation, a straightforward solution to identify them is automated exploit generation [3]. However, it is still an open challenge to have an end-to-end AEG solution for real world large applications like the kernel, due to bottlenecks of symbolic execution, automated heap manipulation, etc.

Instead, we take a different approach that follows human experts to construct synthesized exploits, as shown in Figure 3. Experts first reverse engineering the target program to obtain some knowledge (*e.g.,* what is the consequence of certain inputs), then conceive an input (*i.e.,* attack action selection) in their mind. Next, they will assess the program state after processing the input, and check whether it matches the purpose of exploitation (*i.e.,* inference) based on their exploitation experience. If so, they will repeat this process to conceive further inputs; Otherwise, they will step back and try different inputs. This process stops until a working exploit plan (*i.e.,* attack path) is set up, by then the experts will try to generate an exploit to test and modify it if necessary.

Following this paradigm, we propose to construct an expert system, which first builds a knowledge base of the target program and then explores the knowledge base to generate synthesized attack paths. If an object is abused in a synthesized attack path, it will be marked as sensitive.

3 Our solution: AlphaEXP

3.1 Threat model & Assumptions

First, we assume the most commonly enabled mitigations including SMEP, SMAP [14], KPTI [15], and KASLR [17] are deployed by Linux kernel. Second, we assume the adversary only has a single vulnerability, and our approach only identifies objects that are useful for the exploit of an individual vulnerability rather than exploits involving multiple vulnerabilities. Third, we only generate synthesized attack paths rather than working exploits, and therefore assume the kernel is in an idle state [53] which is easier to assess, and allows our system to focus on sensitive object assessment without considering the reliability of exploitation. Finally, we assume the adversary has the ability to manipulate the heap layout required by the attack path.

3.2 Overview

We follow expert experience to identify sensitive objects. Firstly, experts have to be familiar with the kernel objects, so as to select sensitive ones from them. Secondly, they infer which objects are sensitive, according to the characteristics of the kernel and the experience of exploitation. Finally, they classify those objects based on the feasibility and impact of the associated candidate exploits.

Inspiring by knowledge engineering [44], we present AlphaEXP to simulate the above process and identify sensitive kernel objects. As shown in Figure 4, AlphaEXP consists of three modules: *knowledge graph construction*, *attack path generation*, and *sensitive object classification*. The *knowledge graph construction* module gathers information of the structures and usages of objects in memory, making AlphaEXP familiar with kernel objects. The *attack path generation* module infers which objects can be applied to exploitation, so as to determine which objects are sensitive. The *sensitive object classification* module classifies sensitive objects based on the applied conditions to abuse the objects and the effects achieved of capability upgrades.

3.3 Knowledge Graph Construction

In order to imitate the expert, which has the target program knowledge, a knowledge base is needed for expert system. Since kernel data has many types and many relationships, we choose knowledge graph [37] to store the knowledge. To construct knowledge graph, there are two major stages: *ontology construction* and *knowledge extraction*.

3.3.1 Ontology Construction

To build the knowledge graph, we need to construct the ontology first, which represents heterogeneous knowledge for kernel code (*i.e.,* the ontology encompasses the categories, properties, and relations between the kernel objects, kernel functionalities, and inputs). According to the sensitive object characteristics described in Section 2.3, the ontology should reflect the relationship between userland operations (*i.e.,* inputs) and kernel functionalities, so we design the entities and relations in the ontology based on objects and functionalities, as shown in Figure 5.

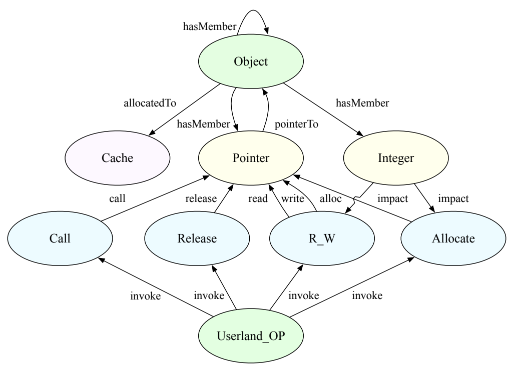


Figure 5: The ontology of knowledge base.

Since objects play an important role in the exploitation, we have an `Object` entity in the ontology. An exploit is often strongly related to the location and layout of the object, therefore we introduce location- and layout-related entities. Regarding the location, a kernel object is often allocated from slab caches, therefore we introduce the `Cache` entity. Regarding the layout, objects may have member variables, among which variables of the `Pointer` and `Integer` type are crucial for exploitation. For instance, variables of the pointer type may be the parameter of memory release functions, and integer variables may be the parameter of memory allocation functions. Therefore, we introduce the `Pointer` and `Integer` entities to the ontology.

For kernel functionalities, we introduce the `R_W`, `Call`, `Release` and `Allocate` entities, as shown in Figure 5. Reading/writing and executing are common operations on sensitive objects, playing an important role in exploitation, therefore we have the `R_W` and `Call` entities. In addition, allocating and releasing of objects are also crucial to exploitaiton. Therefore, we introduce corresponding `Release` and `Allocate` entities in the ontology.

Finally, the knowledge graph should record which userland operations can invoke these kernel functionalities. Because those userland operations are attack actions we can carry out, we need to infer the effects of these actions. `Userland_OP` in the Figure 5 is the embodiment of the above.

3.3.2 Knowledge Extraction

Knowledge extraction is the process of obtaining knowledge from the kernel and organizing it according to the ontology to build the knowledge base. As shown in Figure 6, it consists of two steps: *static knowledge extraction* and *dynamic knowledge extraction*.

Static knowledge extraction In the open source Linux kernel, most knowledge can be obtained through *static analysis*, such as the object's structure and size, and could be further used to build entities and relationships in the knowledge graph, except for those related to `Userland_OP`.

First, we translate the kernel code into LLVM [30] IR (Intermediate Representation) and extract information about the kernel objects and their structure from the IR.

Then, we extract the knowledge about the kernel functionalities. There are many memory management related APIs [46] provided by the kernel (*e.g.,* `copy_from_user()`, `copy_to_user()`, `kmalloc()`, `kfree()`), that are often used by other functions to implement read, write, allocate and release functionalities. For allocating APIs, we check whether the type of the return variables is object and record it in the knowledge graph if so.

Further, we can determine which objects affect the parameters of kernel functionalities by tracing back these use-def chains of the APIs' parameters. Unlike the regular use-def chain tracing, we do not aim to trace back all dependencies of the parameters. Instead, we focus on identifying whether arguments come from an object (or its member variables), and record that object in the knowledge graph if so. We do not need to search for the original source of the parameter, but only the nearest source object, since the knowledge graph contains the relationships between objects which can be indexed to retrieve the original source object. Specifically, for writing, reading, and releasing APIs, we track the use-def chains of their parameters and determine the type of each definition point. If a definition of the object type is found, we stop and record it, otherwise we further track its definition source.

Most of these analyses can be done in intra-procedural. The inter-procedural analysis is only required when the definition point is at the function parameter (*e.g.,* `vmw_execbuf_cmdbuf(..., void *kernel_commands)` calls `copy_from_user(kernel_commands, ...)`), in which case we trace back to the upper level function.

Aliasing can also impact our analysis and may lead to false negatives and false positives. For instance, some alias pointers may point to the same memory pointed by a member pointer of an object, and operations (e.g., read or write) on the alias pointers may affect the use-def chains related to the object, which may make the object sensitive to exploitation. To reduce such false negatives, we use the *AliasAnalysis* pass provided by LLVM to perform alias analysis and recognize such alias pointers. However, an imprecise alias analysis may introduce non-alias pointers and cause false positives to sensitive object identification. So, we only use *MustAlias pointers*, rather than *PartialAlias* or *MayAlias* ones.

In addition, we focus on every `call` instructions, which call function pointers, in LLVM IR (*e.g.,* `call i64 %1(i8* %2, i8* %2)`). By tracking the use-def chain of the function pointer being invoked, we will record it in the knowledge graph if we find that it originates from an object.

Dynamic knowledge extraction This step is to extract the relationship between userland operations and kernel functionalities, and the relationship can be represented as which kernel

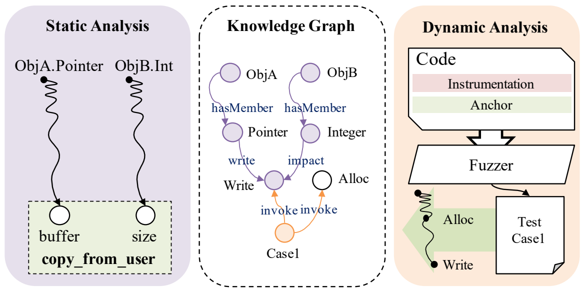


Figure 6: The process of knowledge extraction.

functionalities will be triggered by which userland operation. The knowledge extracted in this part can fill the `UserlandOP` and its related relationships. Since the input construction is difficult for static methods, we take fuzzing to generate the input, and analyze it by dynamic analysis.

We first instrument the kernel source code at the APIs of kernel functionalities to monitor whether they are executed. Then, we utilize fuzzing to generate test cases, which consist of userland operations that invoke the target functionalities. Finally, we save the test cases as `UserlandOP` entities in the knowledge graph and build the relationship with the relevant functionalities, as shown in Figure 6.

3.4 Attack Path Generation

As soon as we collect enough information to represent the facts of the kernel, we consider exploring the knowledge graph to infer potential attack paths for given vulnerabilities.

To generate an attack path, AlphaEXP requires three components: *Attack Action Selection*, which selects a userland operation input to the kernel; *Inference*, which infers the effectiveness of the userland operation in the kernel; and *Scheduling*, which determines whether the inference result is as expected, and also decides whether to add an attack action to the attack path.

3.4.1 Attack Action Selection

Attack actions are the userland operations invoked by the adversary to achieve unintended capability upgrades. All of them are `UserlandOPs` collected in Knowlege Extraction (Sec 3.3.2).

AlphaEXP cannot determine what the most appropriate action is at the moment without inference, because the effect of the attack path is holistic and the choice of attack action is local. Therefore, AlphaEXP adopts a random selection of attack action with heuristics that can reduce the selection range, and leave the task of ensuring the effectiveness of the attack path to scheduling.

AlphaEXP prioritizes actions based on two factors: kmem-cache and the current unintended capabilities. First, the kmem-cache often determines the area where objects are allocated. AlphaEXP prioritizes actions that may yield or affect objects located in the same kmem-cache as objects allocated in the existing (partial) attack path. Second, if the current unintended capability has already reached its maximum strength, then AlphaEXP deprioritizes actions that are only relevant to kernel functionalities with that capability. For example, when current unintended capabilities allow adversaries to read arbitrary addresses, then AlphaEXP stops selecting actions (i.e., `UserlandOPs`) that only involve the "Reading" functionality.

3.4.2 Inference

Given a selected attack action, AlphaEXP then explores the knowledge graph to infer the effect of the action in kernel. Specifically, we implement automated reasoning based on Datalog [8], which could deduce conclusions from facts and rules. Apart from *schemas* and *rules* [41] in our deductive systems [25], we additionally introduce a *framework* for inference.

Schemas We have three types of schemas in our Datalog problem setting: *knowledge graph*, *memory state* and *unintended capability*, as shown in Table 1.

The schemas of *knowledge graph* are all from the relations of the knowledge graph. For example, `CacheOf` describes the `belongsTo` relationship between `Object` and `Cache`.

The schemas of *memory state* are designed to infer the resulting memory state that will arise when kernel executes userland operations. Our emphasis lies on pointers and memory, as they provide a comprehensive description of the memory state. Each pointer and memory possesses its own status, and there exists an intricate interplay between pointers and memory. Therefore, we design various schemas including the `PointTo`, `PointerType`, `PointerStatus`, etc. It is worth noting that `DestroyPointer` is a particular schema designed to distinguish different memory release methods. We define two types of memory release: with and without vulnerability involved. In some cases, the memory release operation may have an invalid (e.g., dangled) pointer as argument and involves a vulnerability, and `DestroyPointer` will not be reasoned. In the most common cases, the memory release has no vulnerability, and `DestroyPointer` can be used to infer further facts.

The schemas of *unintended capability* aim to deduce the unintended capabilities of the current inference state. We categorize these capabilities into three types: reading, writing, and execution. Each class of capabilities is described in its parameters and the way of invoking, with the more special ones being the arbitrary address read (AAR), arbitrary address write (AAW), and arbitrary code execute (ACE). Those capabilities are more powerful and can already do *Arbitrary*, so we only describe the way of invoking.

Rules We design two sets of rules for the categories of *Memory* and *Capability*, as illustrated in Table 1. *Memory* rules are developed based on the allocation of kernel memory. While they do not comprehensively depict the intricate memory management mechanism of the kernel, we believe they are

Table 1: List of schemas and rules in the inference engine.

Schemas	**Knowledge Graph**	**UserlandOP(op: symbol)**	**Performing the userland operation *op*, and each inference needs to start with a *UserlandOP*.**
		CacheOf(obj: symbol,cache: symbol)	The kmem-cache of *obj* is *cache*.
		Invoke(op: symbol, func: symbol)	The userland operation *op* can invoke functionality *func*.
		MemberOf(obj:symbol,member:symbol,offset:number)	The object *obj* has the member *member* at offset *offset*.
		CanWrite(func:symbol,ptr:symbol,size:number)	The functionality *func* can write to the pointer *ptr* with size *size*.
			... ...
	Memory State	DestroyPointer(ptr: symbol)	Destroying the pointer *ptr*. (e.g., *ptr*=0;)
		PointTo(ptr: symbol, mem: symbol)	The pointer *ptr* point the memory *mem*.
		PointerType(ptr: symbol, obj: symbol)	The type of pointer *ptr* is *obj*.
		PointerStatus(ptr: symbol, status: symbol)	The status of pointer *ptr* is *status*, and *status* should be alived or dead.
		MemStatus(mem: symbol, status: symbol)	The status of memory *mem* is *status*, and *status* should be inuse or freed.
		Occupy(mem: symbol, obj: symbol)	The object *obj* is allocated to memory *mem*.
			... ...
	Unintended Capability	NewWrite(func:symbol,ptr:symbol,size:number)	Generated new writing capability *func*, which can write content with length *size* to pointer *ptr*, and it can be directly inferred to produce a new *CanWrite*.
		AAW(func1:symbol,obj1:symbol,func2:symbol,obj2:symbol)	The Object *obj1* can modify the pointer of object *obj2* through function *func1*, enabling arbitrary address writing through function *func2*.
			... ...
Rules	**Memory**	Occupy(M,T) :- MemStatus(M,"freed"),MemCacheOf(M,C),CacheOf(T,C),CanAlloc(Y,T,L,M2,P2),UserlandOP(X),Invoke(X,Y).	
		PointerStatus(P,"alive") :- CanAlloc(Y,O,S,M,P),UserlandOP(X),Invoke(X,Y).	
		PointTo(P,M) :- CanAlloc(Y,O,S,M,P),UserlandOP(X),Invoke(X,Y).	
		PointTo(P,M) :- PointTo(P,M2),Occupy(M2,O),CanAlloc(Y,O,S,M,P),UserlandOP(X),Invoke(X,Y).	
		MemStatus(M,"inuse") :- CanAlloc(Y,O,S,M,P),UserlandOP(X),Invoke(X,Y).	
		MemStatus(M,"inuse") :- Occupy(M,T).	
		... ...	
	Capability	NewWrite(N,NP,Ns) :- CanWrite(N,P,Ns),PointTo(P,M),PointTo(NP,M).	
		AAW(N1,P1,N2,P2) :- CanWrite(N1,P1,Ns),MemberOf(P1,P2,offset),Ns>offset,CanWrite(N2,P2,Ns).	
		... ...	

adequate for inferring attack paths. As for *Capability* rules, they are formulated based on our experiences of exploitation. For example, when the pointer with the write functionality is contaminated, then it may cause arbitrary address write, which is summarized in the practice of exploit.

It is worth mentioning that since we have many rounds of inference, most of our facts are extensional database (EDB), and we will input the results of the previous round in the next round. Therefore, to prevent inference from not converging, our rules all have such a subgoal, which is `UserlandOP` or is inferred from `UserlandOP`. This way, we ensure that the inference converges with only one new `UserlandOP` input.

Framework Our goal is inferring the effect of attack actions, but directly applying Datalog is not feasible. Datalog cannot handle contradictory facts, yet this often happens during reasoning of memory state. For example, assuming the current status is `MemStatus(A,"inuse")` and we enter a `UserlandOP` that releases `A`, then we should infer that `MemStatus(A,"freed")`. Obviously, `MemStatus(A,"inuse")` and `MemStatus(A,"freed")` are contradictory, but Datalog cannot handle the situation, leading to the existence of contradictory facts in the inference state. In this case, the reasoning of the unintended capability will be problematic.

Therefore, we design a two-level framework to address this issue. The first-level engine reasons about the memory state, and the second-level reasons about the capability. After the first level of inference, the memory state will get updated or even become contradictory. Then, we add a *fact adjusting* stage between the two engines, which works as follows:

- The status of memory and pointers are based on the result of the latest operation, overriding previous status.
- Once the pointer state is *dead*, the associated pointing relationship is removed.
- If a pointer points to two memory regions, then these two memory regions must be the same (e.g., a memory is reallocated after freed). We create a new memory entity to unify the two memory regions, and make the pointer point to the new memory entity.

3.4.3 Scheduling

After attack action selection and inference, multiple attack sub-paths will be generated, and AlphaEXP will evaluate them and repeat the process until it finds a suitable attack sub-path that leads to exploitation. AlphaEXP schedules the input for the attack action selection (*i.e.,* along which attack sub-path to select) based on the inference results, and generates the new attack sub-path for scheduling. This cycle continues until an attack path emerges that satisfies the exploitation.

The workflow is shown in Algorithm 1. `Choose_State` selects states based on probability; `Cala_CapabilitiesGap` calculates the gap with exit capabilities; `Get_Action` is the *Attack Action Selection*, and `Infer_Capability` performs effect inference through the *Inference*.

Specifically, we set up a state pool to store the result of each *inference state*. An *inference state* contains information about which attack actions were performed (*i.e.,* attack sub-path), the current unintended capabilities, and the effects of these attacks in the kernel.

From the state pool, AlphaEXP will choose one state and then select a new attack action, inferring its effect for updating the state. No matter what the effect is, AlphaEXP will add the new state to the state pool, simulating the extensive projections in the mind of the exploit expert.

A vital issue in the above process is how to choose the state. As described in Section 2.3, exploitation should be done along the direction of capability upgrading. Therefore, it is preferable to choose the state with the strongest unintended capabilities, but this may lead to a local optimum. Since

Algorithm 1: Workflow of Scheduleing

```
input  : $State_{vul}$, $Set_{action}$, $Cap_{target}$
output : $Path_{attack}$
set $Path_{attack} = \emptyset$ ;
set $Set_{state} = \emptyset$ ;
for $i = Set_{action}.begin;\ i\ != Set_{action}.end;\ ++i$ do
    set $state$ = Copy_State($State_{vul}$);
    append $i$ to $state.path$;
    $state.Length \leftarrow 1$;
    $state.Cap \leftarrow$ Infer_Capability($state.path$) ;
    $state.Value \leftarrow$ Calc_CapabilitiesGap(($Cap_{target}$ , $state.Cap$)) ;
    append $state$ to $Set_{state}$;
end
while $True$ do
    set $CurrentState$ = Choose_State($Set_{state}$) ;
    set $Action_{new}$ = Get_Action($Set_{action}$);
    set $NewState$ = Copy_State($CurrentState$);
    append $Action_{new}$ to $NewState.path$;
    set $NewCap$ = Infer_Capability($NewState$) ;
    if $NewCap > NewState.Cap$ then
        $NewPath.Length \leftarrow 0$;
        $NewState.Value \leftarrow$ Calc_CapabilitiesGap(($Cap_{target}$ ,
          $NewCap$)) ;
    else
        $NewPath.Length ++$;
        $NewState.Value \leftarrow NewState.Value\ /\ NewState.Length$;
    end
    $NewState.Cap \leftarrow NewCap$;
    append $NewState$ to $Set_{path}$;
    if $NewCap >= Cap_{target}$ then
        $Path_{Attack} \leftarrow NewState.path$;
        return $Path_{Attack}$
    end
end
```

some exploits require multiple actions before the effect can be shown, the state with the stronger unintended capability in a single step is not necessarily better than the other state.

Therefore, we set a value for each state to indicate the probability of being selected, and the value is updated according to the following principles.

① **The initial value indicates the gap between the current and the exit capabilities.** There are two types of exit capabilities, both of which require an unintended read capability, as well as an AAW or ACE capability. Therefore, the gap is calculated from two dimensions: one is reading capability, and the other is writing and execution capability.

AlphaEXP assigns a score of one for each dimension, and the gap is the distance between the score and the value of two. The specific scoring is shown in Table 2.

② **Larger with increased unintended capability.** If the unintended capability of the state has been upgraded, the probability of success along this attack path increases. Consequently, the probability of that state being chosen should also increase.

③ **Smaller without increased unintended capability.** Conversely, states without upgraded unintended capability should have a smaller probability of being selected. But we should not give up the state either, and once the unintended ability of the state is subsequently upgraded, its probability will also be significantly increased.

④ **The magnitude of the reduction is proportional to the length of the attack subpath, where no capacity upgrading has occurred.** Generally, as the exploit process advances, fewer actions are needed to enhance the unintended capability, because the difficulty of upgrading the ability will get lower as the capability gets stronger. Therefore, we believe that as the length of the attack subpath, where no capacity upgrading has occurred, increases, we should give it a heavier penalty when it fails to upgrade unintended capability again.

Table 2: List of values for capabilities.

Type	Parameter	Value	Description
Reading	Length Reading Pointer	(0,1)	Any one of the parameters is controlled
Writing	Length	(0,0.125,0.25, 0.375,0.5)	Number of bytes of the control length parameter*
	Writing Pointer	(0,1)	The parameter is controlled
Execution	Pointer	(0,1)	The parameter is controlled

*: Length parameter always occupies four bytes (even if it occupies 8 bytes, the length of 4 bytes of control is enough to complete the goal of the level).

3.5 Sensitive Object Classification

With limited defense resources, objects with high sensitivity levels should be protected as a priority, so sensitive objects should be classified into different levels.

The object's sensitivity depends on how useful it is in exploitation, and this practicality is reflected in: the difficulty of its application and the extent of its effect. Therefore, the sensitive object classification module analyzes many attack paths and classifies the sensitive object used in paths based on *applied conditions* and *effect achieved*.

The factor of the applied conditions is decisive for the sensitivity classification. Objects with high requirements for applied conditions will not be very high in sensitivity, even if the effect achieved is well. It is difficult to upgrade entry capabilities to exit capabilities via a single action, due to the characteristics of sensitive objects and requirement of exit capabilities (as described in Section 2.3). Instead, whether it is easy to apply in vulnerability exploitation becomes the focus.

3.5.1 Applied Conditions Assessment

Applied conditions are divided into three factors: requirements for the kmem-cache, requirements for the entry capa-

Table 3: List of classification indicators.

Perspective	Factor	Description
Applied Conditions	kmem-cache	Can be applied in the exploitation of different vulnerability object memory kmem-cache.
	entry capability	Modification of sensitive object requires unintended writing capability over 0x80 size
	vulnerability type	Sensitive object can both be applied in the exploitation of overflows and UAF
Effect Achieved	writing capability	Sensitive object can be used to upgrade writing capability in exploitation
	executing capability	Sensitive object can be used to upgrade executing capability in exploitation
	reading capability	Sensitive object can be used to upgrade reading capability in exploitation

bility, and requirements for the type of vulnerability, as shown in Table 3.

In exploit development, some objects can only be allocated to a fixed kmem-cache, while some objects can be allocated to different kmem-cache for their elastic attributes. For example, `sembuf` in `do_semtimedop()` is allocated with fix size, so it can only be applied to the scenario when the kmem-cache of vulnerability object is the same as its. In contrast, `drm_property_blob` is allocated with the user-controllable size, so it can be applied to various scenarios with different kmem-cache vulnerability objects.

Some objects require a strong entry capability, while others are not so demanding. For example, to exploit via `tty_struct`, we need to overflow to its 41st member, while via `seq_operations`, we only need to overflow to its first member. Objects with small overflow length requirements and less pressure on their data faking (only the data above the target pointer needs to be faked) tend to be more widely used.

Some objects can only be applied to a fixed type of vulnerability, while others can be applied to various types of vulnerabilities. For example, in the lifecycle of `setxattr()`, allocation and writing are continuous and inseparable, so it is challenging to be corrupted by a buffer overflow vulnerability. Most of the time, it can only be used for UAF to occupy the freed memory.

Of these conditions mentioned above, kmem-cache requirement is the most important, as it distinctly represents the range of possible applications for that sensitive object. Entry capability requirement is second, often determining whether that object is more beneficial. The last is the vulnerability type requirement, because even if an object can be widely applied to a single vulnerability type, it can still be very harmful. However, even if an object can be applied to both UAF and overflow with more demand in terms of cache and capability, its usage is still limited.

3.5.2 Effect Achieved Assessment

The effect achieved is evaluated in factors of the ability to upgrade reading, writing, and execution unintended capabilities, as shown in Table 3.

Some objects can help the exploit development to upgrade both the reading and writing unintended capabilities, such as `msg_msg`, so it can achieve better results. In contrast, some objects can only achieve an effect, and then the object is limited in exploitation.

In terms of the importance of the three unintended capability upgrades, writing unintended capability is most critical, because writing unintended capabilities can often cause the birth of other new unintended capabilities, such as modifying virtual tables and modifying read pointers to produce the reading unintended capability and execution unintended capability. Execution unintended capability is more important than reading unintended capability because execution unintended capability can complete the exploitation through hijacking PC, but reading cannot.

4 Evaluation

The evaluation is designed to answer following questions:

- **RQ1:** How effective is AlphaEXP in sensitive objects identifying and classifying?
- **RQ2:** Is AlphaEXP better at identifying sensitive objects compared to current SOTA techniques?
- **RQ3:** What is the cost of building a knowledge graph?
- **RQ4:** How effective is attack path generation?

4.1 Implementation

We implement a prototype of AlphaEXP based on the technique mentioned above. The Knowledge Graph Construction is developed based on KINT [48] and Syzkaller [19], and consists of 4,120 lines of code in C, Python, and Go. The Attack Path Generation is implemented with Soufflé [28], and consists of 1,170 lines of code.

Regarding dynamic knowledge extraction, we divided the kernel into modules and utilized fuzzing to test each module separately and collect knowledge accordingly. Specifically, we extended Syzkaller's template to cover some missing syscalls and used it to generate test cases. For each module to test, we analyzed the call graphs of the functions within the module, identified all syscalls that might call these functions, and instructed Syzkaller to use the templates related to these syscalls to generate test cases. The kernel is instrumented to monitor memory reading, writing, execution, allocating, and releasing behaviors. During testing, the dynamic knowledge regarding the kernel will be recorded.

The prototype system runs on Ubuntu 20.10 with 128G RAM and Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz. Our kernel evaluation process is performed on the same host and evaluates for kernel v4.15 and v5.5.3 with QEMU [5], which are the same versions as the ones tested by baseline solutions SLAKE [13] and ELOISE [12].

4.2 Experiments Setup

To evaluate our system, we design two sets of experiments, one to evaluate synthesized vulnerabilities, and the other to evaluate real-world vulnerabilities.

Synthesized Vulnerabilities Synthesized vulnerabilities are injected manually, which are summaries of the real vulnerabilities, and are more comprehensive compared to real vulnerabilities for sensitive object assessment. They have different types, different entry capabilities, and different kmem-cache hosting vulnerable objects.

In terms of vulnerability causes, synthesized vulnerabilities include UAF (Use-After-Free) and buffer overflow, which are the two most common types of memory corruption vulnerabilities. In terms of entry capability, synthesized vulnerabilities include: buffer overflows with overrun sizes of 1, 8,

Table 4: List of the sensitive objects, which are identified through the experiments for artificial vulnerabilities by AlphaEXP.

	Sensitive Objects
Write	keyctl_update_key✦, msg_msg✦, add_key✦, ip_options_get_from_user✦, scsi_request, drm_ioctl, __get_filter hiddev_ioctl_usage, proc_ioctl, kexec_segment, do_ipv6_setsockopt, do_semtimedop, vt_do_kdgkb_ioctl, setxattr xt_table_info, gss_pipe_downcall, snd_ctl_elem_info, simple_transaction_argresp, ethtool_set_eeprom, sock_filter proc_do_submiturb, snd_ctl_elem_id, map_lookup_elem, move_addr_to_kernel, compat_agpioc_reserve_wrap drm_syncobj_array_find, fb_sys_write, fb_write, drm_mode_dirtyfb_ioctl, ipv6_txoptions, elf_prpsinfo, sk_buff usblp_write, drm_crtc, drm_property_blob, drm_i915_gem_object, tty_struct, agpioc_reserve_wrap, create_entry drm_syncobj_array_wait_timeout, kernfs_fop_write, kexec_segment, map_update_elem, proc_bulk, sendmsg simple_attr_write, rawv6_seticmpfilter, snd_info_buffer, memfd_create✼, drm_syncobj_timeline_signal_ioctl✼ raw_seticmpfilter✱, cpumask✱
Read	ipv6_opt_hdr★, sock_fprog_kern★, policy_load_memory★, ldt_struct★, ip_options★, cfg80211_wowlan_tcp★ seq_file★, xfrm_policy★, xfrm_algo_aead★, xfrm_algo★, ip_sf_socklist★, proc_dir_entry★, station_info★ cfg80211_pkt_pattern★, user_key_payload★, xfrm_replay_state_esn★, ext4_dir_entry_2★, mon_reader_bin★ sg_header★, tc_cookie★, inotify_event_info★, audit_rule_data★, fb_info★, cfg80211_sched_scan_request★ fb_cmap_user★, cache_request★, fname★, ieee80211_mgd_auth_data★, cfg80211_bss_ies★, cache_reader★ mon_reader_text★, tcp_fastopen_context★, request_key_auth★, xfrm_algo_auth★, cfg80211_scan_request★ msg_msg★, tcp_sock☆, user_element, neighbour, pneigh_entry, net_device, netdev_phys_item_id, rchan_buf netlink_ext_ack, cfg80211_nan_match_params, wiphy, wiphy_iftype_ext_capab, wireless_dev, usb_device, urb hidraw_report, hid_device, sg_request, usblp, drm_crtc, drm_plane, cfg80211_connect_resp_params, fb_cmap beacon_data, probe_resp, cfg80211_roam_info, cfg80211_wowlan_wakeup, cfg80211_ssid, drm_master, seq_buf cfg80211_mgmt_tx_params, ieee80211_mgd_assoc_data, kobj_uevent_env, rpc_pipe_msg, geneve_opt, __kfifo cfg80211_ft_event_params, fat_ioctl_filldir_callback, key_params, drm_property_blob, tcp_fastopen_cookie cfg80211_pmsr_ftm_result✼, cfg80211_update_owe_info✼, cfg80211_fils_resp_params✼, sg_scsi_ioctl✱
Exec	seq_operations✦, perf_event_context✦, linux_binprm✦, vmap_area✦, tty_struct✦, seq_file✦, avc_node✦ kioctx_table✦, snd_seq_timer✦, tty_ldisc✦, sk_security_struct✦, assoc_array_edit✦, cgroup_namespace✦, file✦ ext4_allocation_context✦, tty_file_private✦, subprocess_info✦, timerfd_ctx✦, ccid✦, ip_options✦, kioctx✦ ip_sf_socklist✦, request_key_auth✦, pid_namespace✦, k_itimer✦, ip_mc_list✦, sock✦, ip_mc_socklist✦☆ key✦☆, packet_sock☆, fsnotify_group☆, blk_plug_cb, blk_stat_callback, snd_timer, hci_dev, snd_pcm_runtime drm_i915_gem_object, vga_device, nfs_io_completion, snd_pcm, udp_sock, tracer, snd_timer_instance, dio snd_hwdep, snd_kcontrol, link_master, snd_kctl_ioctl, scsi_cmnd, fbcon_ops, sony_sc, clk_fractional_divider snd_pcm_hw_rule, snd_seq_device, snd_info_entry, snd_card, snd_jack, net_device, pipe_buffer, shm_file_data acpi_cpufreq_data, hid_device, ahci_host_priv, snd_seq_client_port, kprobe, sched_domain_topology_level loop_device, input_polled_dev, hashtab, iommu_group, crypto_ahash, serio, hda_jack_callback, akcipher_instance ahash_request_priv, crypto_tfm, skcipher_instance, journal_s, input_dev_poller, alps_data, nf_conntrack_expect crypto_skcipher, aead_instance, crypto_acomp, ubuf_info, psmouse, ml_device, rpc_task, kthread_create_info proc_inode, proc_dir_entry, fib6_walker, aio_kiocb, simple_attr, inet_connection_sock, nfs_server, ring_buffer async_entry, filter_pred, nfs_renamedata, nfs_commit_data, nfs_pgio_header, hda_codec, rtnl_link, rpc_rqst flow_block_cb✼, flow_indr_block_cb✼, tcf_filter_chain_list_item✼, io_wq✼, execute_cb✼, context_barrier_task✼

✦: Identified by SLAKE as well, ★: Identified by ELOISE as well, ☆: Identified by KOOBE as well
✼: Not present in v4.15, ✱: False Positives

and 1024, UAF with both reading and writing functionalities, UAF with reading functionality, and UAF with writing functionality. These cover almost all potential entry capabilities that adversaries could get. In terms of kmem-cache, the vulnerable objects of synthesized vulnerabilities are allocated to 14 different kmem-cache: kmalloc-8, kmalloc-16, kmalloc-32, kmalloc-64, kmalloc-96, kmalloc-128, kmalloc-256, kmalloc-512, kmalloc-1024, kmalloc-2048, and some special kmem-cache (*e.g.,* seq_file_cache).

In total, we have crafted 84 vulnerabilities and inserted them into crafted vulnerable kernel drivers. Due to the comprehensiveness, experiments on synthesized vulnerabilities can more adequately identify the sensitive object and assist in classifying and evaluating sensitive objects.

Real-world Vulnerabilities We exhaustively search Linux kernel vulnerabilities with public exploits in the recent five years from the CVE security vulnerability database [33]. In total, we collect 19 vulnerabilities, each of which is either a UAF or a heap buffer overflow.

The experiments on real-world vulnerabilities aim to calcu-

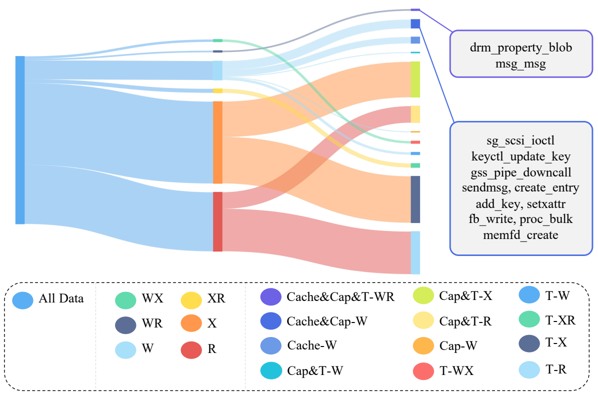


Figure 7: Results of object sensitivity classification.

late the recall of sensitive objects identified by experiments on synthesized vulnerabilities.

4.3 Results of Sensitive Object Assessment

To answer **RQ1**, we conduct experiments on both synthesized vulnerabilities and real-world vulnerabilities. During knowledge graph construction, we tested each kernel module (as described in Section 4.1) for 72 hours to perform dynamic knowledge extraction.

For synthesized vulnerabilities, we perform attack path generation for each set of test cases (i.e., proof-of-concept inputs triggering the vulnerabilities) and extract sensitive objects from them. We have slightly modified the attack path generation, in which we remove the state that already has exit capabilities from the state pool and continue to generate other attack paths until the time threshold (1 hour) is reached. The time threshold was determined by our manual validation. We have also extended the exploration time to 2 hour in our experiment, but found no more attack paths or sensitive objects, and all attack paths and sensitive objects found during the extra time window are repetitive.

Finally, we recognized 50 objects that can be abused to upgrade writing capability, 81 with reading capability, and 112 with execution capability.

The full list of sensitive objects is shown in Table 4. Most of them are present in both kernel versions, except for 11 objects are not present in v4.15, indicating that the newer kernel version could have more sensitive objects and many sensitive objects exist in most versions.

We classify the sensitive object into 12 levels, as shown in Figure 7 (The complete list is in Appendix A.1). The levels are labelled based on both the low requirements of *applied conditions* and the high capability of *effects achieved*. The applied conditions include kmem-cache (`Cache`), entry capability (`Cap`), and the type of vulnerability (`T`). The effects achieved include writing (`W`), reading (`R`), and executing (`X`) unintended capabilities. Different combinations of these symbols indicate the applied conditions and effect achieved of sensitive objects at that level. For example, `Cache&Cap&T-WR` means that the sensitive object can be used in the exploitation has low requirements for kmem-cache, entry capability, and the type of vulnerability, and the sensitive object can upgrade the unintended capabilities of writing and reading in the exploit.

4.3.1 False Positives

An object is sensitive if and only if it could be abused in an attack path and facilitate the exploitation. AlphaEXP reports sensitive objects without real exploit evidence, and therefore may have false positives. However, our goal is not generating working exploits, we therefore have two reasonable assumptions: (1) there is a proper vulnerability that could tamper with the reported object, and (2) a final exploit could be generated as long as we can tamper with the reported object and find an attack path. Therefore, we only need to prove that the reported object could facilitate attack path generation, if it is tampered with a proper synthesized vulnerability.

To achieve this, we utilize a debugger to modify the object in the guest kernel that is running in QEMU, which simulates a synthesized vulnerability. The modification is carried out based on the attack path generated by AlphaEXP . Afterward, we monitor the kernel to confirm whether the expected capability upgrades (as indicated by the object's sensitivity level) have been met. Since this is a dynamic verification, we believe it is accurate.

Following this procedure, we have successfully validated all sensitive objects identified by AlphaEXP, and only found 3 false positives: (1) the `sock` object in `raw_seticmpfilter()`, (2) the `cpumask` object in `get_user_cpu_mask()`, and (3) the `buffer` object in `sg_scsi_ioctl()`.

Regarding the first two false positives, the objects could be abused to write content into kernel buffers. However, the content is sanitized by the kernel, so that the intended unintended capability upgrade fails. Regarding the third false positives, the object could be abused to read kernel content to userland. However, the readout content originates from the userland and is useless for exploitation.

4.3.2 False Negatives

Our prototype may also have false negatives, due to the imperfect techniques we take to build the knowledge graph and search for attack paths, as well as the incomplete list of starting vulnerabilities, which may miss some potential entity or relationships in the knowledge graph or some attack paths. However, we do not have the ground truth of all sensitive objects (could be abused by exploits) and their sensitivity, which is infeasible to get even by experts or via analyzing security patches. Instead, we utilize public exploits that will corrupt sensitive objects to launch attacks as the baseline to evaluate the false negatives of AlphaEXP .

Table 5: List of programs and results evaluated with AlphaEXP .

CVE-ID	Type	Vul. Cap.	Sensitive objects abused by the public exploit	Partial sensitive objects identified by AlphaEXP
CVE-2022-27666	overflow	Overwrite(skcipher_walk,0x1000)	msg_msg, user_key_payload	msg_msg, drm_property_blob, user_key_payload
CVE-2022-25636	overflow	Overwrite(net_device,0x18)	msg_msg, net_device	setxattr, msg_msg, net_device, sendmsg, create_entry, add_key, memfd_create
CVE-2022-0995	overflow	Overwrite(watch_filter,0x2)	msg_msg, sk_buff	pipe_buffer, msg_msg, sk_buff, sendmsg, create_entry, proc_bulk
CVE-2022-0185	overflow	Overwrite(fs_context,0x40)	msg_msg, pipe_buffer	pipe_buffer, msg_msg, sendmsg, create_entry, setxattr, tty_struct
CVE-2021-43267	overflow	Overwrite(tipc_aead_key,0x20)	msg_msg, tty_struct	msg_msg, sendmsg, create_entry, setxattr, tty_struct
CVE-2021-42327	overflow	Overwrite(wr_buf,0x40)	msg_msg, subprocess_info	msg_msg, subprocess_info
CVE-2021-42008	overflow	Overwrite(sixpack,0x1000)	msg_msg, shm_file_data	msg_msg, seq_operations, shm_file_data
CVE-2021-41073	UAF	UAF(kmalloc-32-obj)	setxattr, seq_operations	msg_msg, setxattr, sendmsg, seq_operations, shm_file_data
CVE-2021-3573	UAF	UAF(hci_dev)	setxattr, hci_dev	setxattr, add_key, sendmsg
CVE-2021-32606	UAF	UAF(isotp_sock)	setxattr, sock	setxattr, sendmsg, add_key, sock
CVE-2021-33909	overflow	Overwrite(seq_file,-0x1000)	eBPF	✘
CVE-2021-26708	UAF	UAF(virtio_vsock_sock)	msg_msg, sk_buff	drm_property_blob, msg_msg, sk_buff, pipe_buffer
CVE-2021-22555	UAF	UAF(msg_msg)	msg_msg, sk_buff, pipe_buffer	drm_property_blob, msg_msg, sk_buff, pipe_buffer
CVE-2021-20226	UAF	UAF(files_struct)	setxattr, map_lookup_elem, map_update_elem	drm_property_blob, msg_msg, setxattr, map_lookup_elem, map_update_elem
CVE-2020-27194	overflow	Overwrite(eBPF,0x1000)	eBPF	✘
CVE-2020-14381	UAF	UAF(super_block)	sendmsg, super_block	sendmsg, setxattr, super_block
CVE-2019-18683	UAF	UAF(vb2_buffer)	setxattr, vb2_buffer	setxattr, sendmsg, vb2_buffer
CVE-2019-15666	UAF	UAF(xfrm_policy)	setxattr, xfrm_policy	setxattr, add_key, xfrm_policy
CVE-2018-6555	UAF	UAF(ias_object)	irda_queue, XFRM_socket	✘

Specifically, we first run the public exploits and collect sensitive objects abused by them as the ground truth. Then, we run AlphaEXP on these real world vulnerabilities to infer attack paths and sensitive objects, then calculate the fase negatives. The evaluation result is shown in Table 5. AlphaEXP can generate 16 attack paths for the 19 CVEs, and miss 3 sensitive objects: `eBPF`, `irda_queue` and `XFRM_socket`.

For CVE-2021-33909 and CVE-2020-27194, AlphaEXP fails to analyze advanced features like eBPF and thus cannot recognize the sensitive objects. For CVE-2018-6555, AlphaEXP fails to generate test cases via fuzzing to trigger functionalities related to the `XFRM_socket` object, and `irda_queue` are not appears in kernel v5.5.3, so AlphaEXP cannot generate the attack path.

Further, there are two sensitive objects `super_block` and `vb2_buffer` missing in Table 4, but could be identified by AlphaEXP in Table 5. It indicates that AlphaEXP may also have false negatives due to the list of synthesized vulnerabilities to analyze is incomplete.

4.3.3 Compare with SOTA

To answer **RQ2**, we choose SLAKE [13], KOOBE [11], and ELOISE [12], which are state-of-the-art solutions in kernel sensitive object identification, as the comparison targets.

We can identify all the sensitive objects identified by SLAKE, KOOBE, and ELOISE, as marked in Table 4, and we were able to classify sensitivity levels, which SLAKE and ELOISE did not do.

SLAKE identifies the victim objects and the spray objects. The victim objects are sensitive objects that can help achieve arbitrary code execution, which are all listed in the *Exec* row of Table 4. The spray objects are sensitive objects that are allocated to host data written by userland. Such objects are all listed in the *Write* row of Table 4. AlphaEXP can identify more objects in addition to objects that cause execution or spraying, such as objects that cause arbitrary address writes.

KOOBE mainly focuses on using sensitive objects for exploitation, rather than identifying them. As a result, the authors manually collected some commonly used objects from public exploits and also took some objects from SLAKE. [2]

ELOISE identifies elastic objects, which are used in the reading function and can control the size. Such objects are all listed in the *Read* row of Table 4. AlphaEXP can identify more objects including non-elastic objects, e.g., `seq_file`.

In summary, AlphaEXP is more effective than the state-of-the-art solutions in identifying kernel sensitive objects and can also classify the sensitivity of those objects.

4.3.4 Case Studies

We select two high-ranking sensitive objects from the results to illustrate how they contribute to exploit development and why they are assigned such a high sensitivity. To express the generality of AlphaEXP , we only choose sensitive objects that do not have public exploits to abuse them, rather than well known objects (*e.g.,* `msg_msg` [4]).

The first sensitive object we chose is `drm_property_blob`, which can abused to complete the functionality of writing and reading. The function `drm_mode_createblob_ioctl()` allocates and writes the content from userland to `drm_property_blob`. This object's size is controlled by the user, and the userland content could be filled into its `data` field. In other words, the object can be allocated to multiple kmem-cache and can be used as a heap spraying object and facilitate exploitation for vulnerabilities like UAF. In addition, the function `drm_mode_getblob_ioctl()` can read the content from kernel to userland, while the size and the target memory are all controlled by this object. In other words, abusing this object could upgrade the unintended capability to arbitrary reading.

Another sensitive object we chose is the local object `name`

[2]It is confirmed by private correspondence with the authors of KOOBE.

used in the function `memfd_create()`, which is similar to `msg_msg`. Its allocation length can be controlled by the user, and its parent function can be called directly with the syscall `__NR_memfd_create`, and write the user data to kernel buffer. Therefore, in combination with the `userfaultfd` technique [32], this object is valuable for heap spraying and UAF exploitation.

4.4 Knowledge Graph Construction (RQ3)

The static knowledge extraction process takes 19 minutes to analyze all kernel code and generate a knowledge graph comprising 358,383 entities and 440,741 relationships. The dynamic knowledge extraction process takes 72 hours, mostly for fuzzing as described in Section 4.1. After this process, we removed entities (and their attributes) that could not be directly or indirectly connected to `UserlandOP` entities, leaving 100,723 entities and 180,204 relationships.

4.5 Attack Path Generation (RQ4)

4.5.1 Ablation Study

AlphaEXP utilizes two heuristics in attack path generation: one for attack action selection (§ 3.4.1), and another for state selection in scheduling (§ 3.4.3). To evaluate the effectiveness of these heuristics, we designed four groups of experiments for ablation study: (1) the *baseline* group uses random strategies, (2) the *baseline+AS* group uses the heuristic for attack actions selection, (3) the *baseline+SS* group uses the heuristic for state selection, and (4) the *AlphaEXP* group utilizes both heuristics for attack action and state selection.

We performed 20 experiments for each of the four groups, to generate attack paths for a UAF vulnerability that can be exploited using four key actions (as shown in Appendix A.2). Note that, although we only evaluate this specific UAF vulnerability, this scenario is representative and the results should be consistent. First of all, this scenario is common for most exploitation involving UAF. Second, this scenario is also similar to other exploitations, e.g., those involving buffer overflows, since they in general also rely on proper memory allocation and memory read/write. Lastly, the length of the attack path is this scenario is greater than that of most other scenarios.

For each group of experiment, the time consumption of generating the expected attack action is recorded, as shown in Figure 8. At the early stage, the action selection heuristic shows greater improvements to the performance, since there are only a few states to select from. As the attack path generation process goes on, the state pool grows significantly, and the state selection heuristic thus shows great improvements, because the random strategy can select the appropriate state with low probabilities.

4.5.2 Patterns of the Generated Attack Paths

We analyzed the attack paths generated by AlphaEXP for the 84 *Synthesized Vulnerabilities*, one of which is listed in

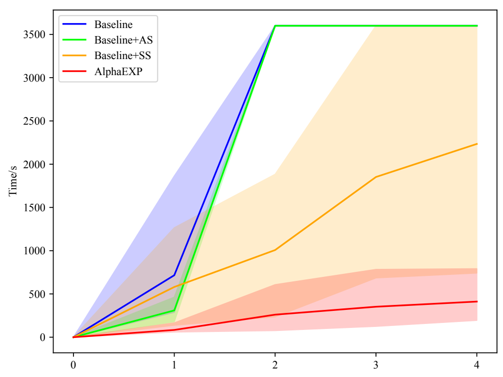


Figure 8: The result of the ablation experiments with heuristics. The vertical axis represents the run time and the horizontal axis represents the key actions in the attack path.

Appendix A.2, and classified their patterns into the following four categories:

① **UAF to AAR**. Assuming there is a UAF vulnerability, where a vulnerable object is freed but still pointed by a dangling pointer, the adversary could follow this type of attack paths to get the capability of arbitrary address read (AAR). In the first case, the vulnerable object has a pointer field which will be read by the UAF. Then, the attack path will guide the adversary to occupy the memory of the vulnerable object (e.g., via heap spraying) with an object with writing functionality. Lastly, the adversary could abuse the writing functionality to control the pointer field of the vulnerable object and cause AAR. In another case, the vulnerable object has no pointer field to be read by UAF. First, the attack path will guide the adversary to first occupy the vulnerable memory with an object with reading functionality. Then, it frees this object and reallocates it to an object with writing functionality. Lastly, the adversary could abuse the writing functionality to control the object with reading functionality and cause AAR.

② **UAF to ACE**. This process is similar to the one above. This type of attack paths guides the adversary to occupy the vulnerable memory with an object with the executing functionality and another object with the writing functionality. Then, the adversary can abuse the writing functionality to control the object with executing functionality (e.g., by tampering with its code pointer field) and cause ACE.

③ **UAF to AAW**. Similar to UAF to AAR, this category has two sub-types of attack paths. In the first case, the vulnerable object has a pointer field which will be write by the UAF. Then, the attack path will guide the adversary to occupy the memory of the vulnerable object with an object with writing functionality. Lastly, the adversary could abuse the writing functionality to control the pointer field of the vulnerable object and cause AAW. In the second case, the adversary could directly occupy the vulnerable memory with an object with a pointer field to write. Then, it frees this object and reallocates it to another object with writing functionality. The adversary could abuse the writing functionality to control the object with writing pointer and cause AAW.

④ **Overflow to reading, execution, and writing capabilities**. Assuming there is a buffer overflow vulnerability, where a vulnerable object is overflowed and following victim will be overwritten, then the adversary could follow this type of attack paths to get the capability of memory reading, writing and execution. In general, the attack paths will guide the adversary to place sensitive objects with reading, writing or executing capabilities after the vulnerable object. Then, the adversary could abuse the overflow vulnerability to overwrite the sensitive objects (e.g., their size field or pointer fields) to upgrade and get the reading, execution, and writing capabilities.

4.5.3 Findings

Based on the analysis, we have two interesting findings:

AAW is more challenging to achieve than ACE. In our experiments, there are more attack paths able to achieve ACE than to achieve AAW. There are few objects in the kernel can be used to achieve AAW. Most kernel objects with writing functionality are allocated to a fixed cache with a non-controllable length, and their pointer fields are not placed at the beginning of objects. In order to tamper with such objects and get AAW, the adversary in general has to either find a buffer overflow vulnerability able to overwrite a large size or find a UAF vulnerability that can allocate objects in the same kmem-cache. On the other hand, kernel objects with execution functionality scatter all over the kernel and can be abused to launch ACE.

UAF vulnerabilities are easier to exploit than overflow vulnerabilities. When generating attack paths, we found that UAF vulnerabilities are more frequently exploited than overflow vulnerabilities, with a higher number of successful exploits. Given a UAF vulnerability, an adversary can have one memory occupied by multiple objects with different functionalities. So, writing one occupying object would cause other occupying objects misbehave and get unintended capabilities. And many UAF vulnerabilities have pointers, which are the target of the reading and writing functionality, thus reducing the difficulty of exploitation. In contrast, an overflow vulnerability can only be exploited by modifying the pointer of an adjacent object to complete an unintended capability upgrade, so it is relatively difficult to exploit.

5 Discussion

Soundness. We tested the sensitive objects identified by AlphaEXP and found that, depending on their usage in the attack path, they can facilitate capability upgrades. However, due to the enormous workload, we were not able to generate exploits for all attack paths, and therefore cannot fully guarantee their effectiveness in real-world exploits. Future work could address the challenge of automating the evaluation of the effectiveness of sensitive data in practical exploitation.

Completeness. AlphaEXP relies on target vulnerabilities to identify sensitive objects. While we have constructed some synthesized vulnerabilities that cover most of the initial capabilities adversaries could obtain (to the best of our knowledge), it is unrealistic to collect all potential vulnerabilities for sensitive object identification. Furthermore, using fuzzing to obtain `Userland_OP` entities may not achieve full coverage. Additionally, the rules we currently have only focus on memory allocation and may not fully reflect the complex memory management mechanism of the kernel (*e.g.,* the list mechanism [38]), potentially missing some objects that have a specific usage in the exploit. We leave it as an future work to design an approach that does not rely on vulnerability input. The fuzzer used by AlphaEXP can also be improved, for example, by using directed greybox fuzzing to achieve higher coverage of the target code area obtained from static analysis. Finally, more complete rules can be designed or implemented for inference in a real world environment.

Knowledge Graph. We have created a knowledge graph for the Linux kernel, but the elements are still expandable, such as whether functionalities are sanitized when they are called. With these elements, a more detailed and reliable blueprint for exploitation can be generated. In addition, we can also apply this idea to other software, such as userland software, to build a knowledge graph of the vulnerable software and generate a plan for exploitation, which can significantly reduce manual work.

Attack Action Selection. Our current selection method can be adapted to the task at hand, but it does not inherit the experience of the previous selection each time it is used. Therefore we can add artificial intelligence algorithms to it in the future to make the method more efficient and accurate.

Automated Exploit Generation. Although we did not need to generate exploits to find sensitive objects, our approach provides a new idea for automated exploit generation (AEG). In fact, AEG relies on PoC inputs or diverging inputs [49] to get the required information, but often overlooks the strategy of experts in exploitation. Experts will first perform a program analysis and then generate an exploit plan based on the vulnerability and their knowledge of the program, which may only appear in their mind. AEG solutions lack support for generating exploit plans or provide the raw materials needed for exploitation. Therefore, we believe it is possible to refer to our approach and refine its components (*e.g.,* heap manipulations) so that AEG can be better applied in practice.

6 Related work

6.1 Sensitive Kernel Object identification

SLAKE [13] and ELOISE [12] are state-of-the-art solutions in identifying sensitive kernel objects. SLAKE identifies the spray object and victim object in the kernel, while ELOISE identifies the elastic object in the kernel. The above structures are all part of the sensitive object and do not further classify the identified object. There is a lack of systematic effort to identify and analyze sensitive objects.

6.2 Automated Exploit Generation

AlphaEXP can also be used in AEG (Automated Exploit Generation), which is still an open challenge. A few number of solutions have been proposed.

6.2.1 End-to-End AEG System

The APEG [7] is an early attempt at AEG based on the vulnerability patch. The AEG [3] solution uses source code assistance to extract the path constraints that trigger the vulnerability. Then many end-to-end systems, binary-based, are presented. Mayhem [9], CRAX [26] and Rex [42] are all based on symbolic execution. These AEG solutions follow a similar workflow, collecting symbolic constraints of the execution trace of PoC by symbolic execution and exploiting based on the exploit pattern of different vulnerabilities. However, all of the binary-based end-to-end systems require exploitable PoCs, which are rare in the real world. Revery [49] attempts to exploit the vulnerability provided with non-exploitable PoCs. It proposes a novel layout-oriented fuzzing and a control-flow stitching solution to explore the exploitable state of the target program. Tunter [47] improves the efficiency of exploitable state exploration by using a taint-guide approach.

The above-mentioned work pushed the development of AEG, but none of them jumped out of two limitations: 1) no attempt was made to obtain more information needed for exploit from analysis binary; 2) the conditions needed for exploit were framed, and the system could only work when the conditions were all met.

6.2.2 Heap Manipulation in AEG

The exploitation of heap-related vulnerabilities requires the specific heap layout, which is called heap feng shui [43]. Therefore, several works aiming at automated heap manipulation were proposed. SHRIKE [22] proposed a method of constructing the input sequence and laying out the heap memory by analyzing the PHP heap management, Gollum [23] uses a pure gray box method, which does not require symbolic execution or white-box analysis. Instead, it performs lightweight fuzzing. SLAKE [13] proposes a technique on the kernel to manipulate the slab layout. Maze [50] models the problem as a Linear Diophantine Equation and solves it deterministically.

6.2.3 Mitigation Bypassing in AEG

Mitigation is a security mechanism to raise the threshold of exploitation, and have applied in modern operating systems. Therefore, adversary should bypass the mitigation for exploit generation. Q [40] is devoted to bypassing DEP [2]. [10, 18, 24, 27, 39] present systematical analysis on CFI bypassing.

6.2.4 Expert System in AEG

Expert system is a possible future direction for AEG. So far, no related works have been studied on this topic. An expert system can generate an exploit plan and guide the exploit. It can help expand the scope of the application of AEG. We can generate a suitable exploit plan based on the current knowledge of the target program, rather than first developing a good pattern of exploitation, and then finding the suitable programs. Therefore, an expert system that has knowledge of exploitation, can acquire knowledge of the target program, and generate the plan, is needed. AlphaEXP is a preliminary attempt at expert system, which is obviously not yet so powerful but suggests a possible idea.

7 Conclusion

In this paper, we present the first expert system AlphaEXP for identifying sensitive kernel objects and classifying their sensitivity. This system works by following experts to build a knowledge graph of the kernel and then explores the graph to build synthesized attack paths for given vulnerabilities. Results showed that, this system could help identify a large number of sensitive kernel objects which are worthy to protect. AlphaEXP also provides a possible future direction for AEG.

Acknowledgments

We sincerely thank all the anonymous reviewers and the shepherd for their valuable comments and suggestions to help us improve this work. This work is supported in part by the National Key Research and Development Program of China (2021YFB3100500, 2021YFB2701000), China Postdoctoral Science Foundation (2021M701942), and National Natural Science Foundation of China (61972224).

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] S. Andersen and V. Abella. Memory protection technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[4] Markel Azpeitia Loiti. Gaining root access in linux using the cve-2021-26708 vulnerability. 2021.

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[6] Eloi Benoist-Vanderbeken and Fabien Perigaud. An apple a day keeps the exploiter away.

[7] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 143–157. IEEE, 2008.

[8] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[10] Kaixiang Chen, Chao Zhang, Tingting Yin, Xingman Chen, and Lei Zhao. VScape: Assessing and escaping virtual call protections. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1719–1736, 2021.

[11] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1093–1110, 2020.

[12] Y. Chen, Z. Lin, and X. Xing. A systematic study of elastic objects in kernel exploitation. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[13] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1707–1722, 2019.

[14] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/, 2012.

[15] Jonathan Corbet. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/, 2017.

[16] Thomas Dullien and Halvar Flake. Exploitation and state machines. *Proceedings of Infiltrate*, 2011.

[17] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/, 2013.

[18] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[19] Google. Syzkaller. https://github.com/google/syzkaller.

[20] Samuel Groß. Thursday, january 9, 2020 remote iphone exploitation part 3: From memory corruption to javascript and back – gaining code execution. https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html.

[21] Haxxin. Exploiting cve-2021-43267. https://haxx.in/posts/pwning-tipc/, 2021.

[22] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, 2018.

[23] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1689–1706, 2019.

[24] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

[25] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216, 2011.

[26] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 78–87. IEEE, 2012.

[27] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882, 2018.

[28] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[29] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, 2014.

[30] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[31] S. H. Liao. Expert system methodologies and applications - a decade review from 1995 to 2004. *Expert Systems with Application*, (1):28, 2005.

[32] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1963–1976, 2022.

[33] MITRE. Cve. https://cve.mitre.org/.

[34] MITRE. Cve-2021-43267. https://www.cvedetails.com/cve/CVE-2021-43267.

[35] PaX-Team. Pax aslr (address space layout randomization). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[36] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[37] Jay Pujara, Hui Miao, Lise Getoor, and William Cohen. Knowledge graph identification. In *International semantic web conference*, pages 542–557. Springer, 2013.

[38] Randorisec. Cve-2022-34918-lpe-poc. https://github.com/randorisec/CVE-2022-34918-LPE-PoC.

[39] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[40] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, volume 10, 2011.

[41] Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers Inc., 2002.

[42] Shellphish. Rex. https://github.com/angr/rex, 2015.

[43] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007:11–20, 2007.

[44] Rudi Studer, V.Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25(1):161–197, 1998.

[45] Inc. Qualcomm Technologies. Pointer authentication on armv8.3: Design and analysis of the new software security instructions. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[46] the kernel development community. Memory management apis. https://www.kernel.org/doc/html/latest/core-api/mm-api.html.

[47] Ruipeng Wang, Kaixiang Chen, Zulie Pan, Yuwei Li, Qianyu Li, Yang Li, Min Zhang, and Chao Zhang. Tunter: Assessing exploitability of vulnerabilities with taint-guided exploitable states exploration. *Computers & Security*, 124:102995, 2023.

[48] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, 2012.

[49] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1914–1927, 2018.

[50] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1647–1664, 2021.

[51] Ned Williamson. Sockpuppet: A walkthrough of a kernel exploit for ios 12.4. https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html.

[52] willsroot. corctf 2021 fire of salvation writeup: Utilizing msg_msg objects for arbitrary read and arbitrary write in the linux kernel. https://www.willsroot.io/2021/08/corctf-2021-fire-of-salvation-writeup.html.

[53] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, 2022.

A Appendix

A.1 Sensitivity Classification

A complete sensitivity classification ranking of the sensitive objects is shown in the Table 6. There are also sensitive objects that are not included in the table, because they do not meet the requirements of `Cache`, `Cap` and `T`.

A.2 Case Study of the Generated Attack Path

We choose an attack path generated by AlphaEXP, to show the process of its generation. Figure 9 illustrates the attack path of a vulnerability, which is one of synthesized vulnerabilities, and how to generate the path.

The vulnerability is an UAF vulnerability with reading functionality, and the vulnerable object will be allocated to kmalloc-32. Figure 9 (b) illustrates the attack path of the vulnerability, and Figure 9 (c) illustrates the part of the knowledge graph, which contains the elements required for this attack path generation. It first invokes `OP-21`, which causes sensitive object `setxattr-ana-88` to be allocated into the vulnerable memory, resulting in the two type pointer point to the same memory, and the writing functionality can tamper the `pointer-v`, the reading pointer, to complete the AAR, as shown in Figure 9 (a). Then, the UAF is triggered again, by invoking `OP-v1`, and through invoking `OP-328`, similar to the above process, `seq_operations` will be allocated into the UAF memory. Next, repeat the above process, using `OP-21` to allocate the `setxattr-ana-88` into the UAF memory, and the writing functionality can tamper the function pointer of `seq_operations`, as shown in Figure 9 (a). In the above process, new capability ACE are acquired, and capability upgrades are completed.

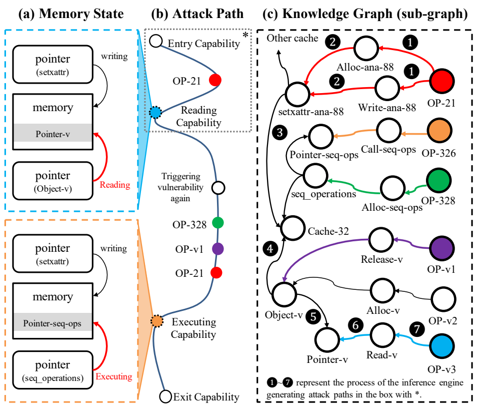


Figure 9: Example of attack path generation.

In addition, we select a portion of the attack path generation (as the box marked with an * in Figure 9 (b)) to show how the inference engine works. When invoking the `OP-21`, the engine infer that the functionalities of `Alloc-ana-88` and `Write-ana-88`, as ❶. Then the engine infer the target of those functionalities is `setxattr-ana-88`, as ❷, and find it can be allocated to `Cache-32`, which is the kmem-cache of vulnerable object memory, as ❸ and ❹. Next, memory allocated can occupy the vulnerable memory due to the UAF, and there is a reading functionality, invoked by `OP-v3` can readout from the pointer in this memory, as ❺ to ❼. Since the same memory is pointed to by two pointers, and one contains a read pointer and the other has a write function, the engine infers that a new read capability is created.

Table 6: List of object sensitivity classification.

Cache&Cap&T-WR	drm_property_blob, msg_msg
Cache&Cap-W	sg_scsi_ioctl, keyctl_update_key, sendmsg, gss_pipe_downcall, create_entry, add_key, setxattr, fd_write, proc_bulk, memfd_create
Cache-W	xt_table_info, simple_attr_write, proc_do_submiturb, usblp_write, ip_options_get_from_user, ipv6_txoptions, do_ipv6_setsockopt, sk_buff
Cap&T-W	do_semtimedop
Cap&T-X	seq_operations, assoc_array_edit, cgroup_namespace, ext4_allocation_context, ip_options_rcu, ip_sf_socklist, pid_namespace, avc_node, tty_ldisc tty_file_private, file, ccid, blk_plug_cb, snd_timer_instance, link_master, snd_info_entry, hda_jack_callback, hashtab, shm_file_data, crypto_ahash ahash_request_priv, crypto_tfm, skcipher_instance, akcipher_instance, crypto_skcipher, aead_instance, crypto_acomp, ubuf_info, flow_block_cb, rpc_task kthread_create_info, tracer, nfs_io_completion, io_wq, simple_attr, input_polled_dev, input_dev_poller, acpi_cpufreq_data, clk_fractional_divider fbcon_ops, pipe_buffer, ip_mc_socklist
Cap&T-R	user_element, request_key_auth, user_key_payload, seq_buf, pneigh_entry, netdev_phys_item_id, tc_cookie, cfg80211_nan_match_params wiphy_iftype_ext_capab, cfg80211_connect_resp_params, cfg80211_fils_resp_params, cfg80211_roam_info, cfg80211_ssid, cfg80211_update_owe_info key_params, cfg80211_pkt_pattern, cache_request, tcp_fastopen_cookie, beacon_data, fat_ioctl_filldir_callback, hidraw_report
Cap-W	snd_info_buffer
T-WX	tty_struct, ip_options, drm_i915_gem_object
T-W	rawv6_seticmpfilter, kernfs_fop_write, fb_sys_write
T-XR	seq_file,ip_sf_socklist, net_device, hid_device
T-X	perf_event_context, linux_binprm, vmap_area, kioctx_table, kioctx, ip_mc_list, k_itimer, sk_security_struct, snd_seq_timer, timerfd_ctx, subprocess_info key, sock, blk_stat_callback, snd_timer, snd_pcm_runtime, snd_pcm, snd_hwdep, snd_kcontrol, snd_kctl_ioctl, snd_pcm_hw_rule, snd_seq_device, snd_card snd_jack, snd_seq_client_port, hda_codec, kprobe, nf_conntrack_expect, rtnl_link, flow_indr_block_cb, tcf_filter_chain_list_item, fib6_walker inet_connection_sock, packet_sock, rpc_rqst, hci_dev, udp_sock, sched_domain_topology_level, async_entry, ring_buffer, filter_pred, nfs_renamedata nfs_server, nfs_pgio_header, nfs_commit_data, proc_inode, proc_dir_entry, aio_kiocb, dio, journal_s, serio, ml_device, alps_data, psmouse, iommu_group loop_device, sony_sc, ahci_host_priv, scsi_cmnd, nvmem_device, vga_device, context_barrier_task, execute_cb
T-R	ipv6_opt_hdr, sock_fprog_kern, policy_load_memory, ldt_struct, ip_options, xfrm_replay_state_esn, cache_reader, cfg80211_bss_ies, sg_header inotify_event_info, fb_cmap_user, fname, ieee80211_mgd_auth_data, tcp_fastopen_context, xfrm_algo_auth, cfg80211_wowlan_tcp, xfrm_algo xfrm_algo_aead, cfg80211_scan_request, mon_reader_bin, cfg80211_sched_scan_request, mon_reader_text, station_info, ext4_dir_entry_2, xfrm_policy fb_info, audit_rule_data, n_tty_data, proc_dir_entry, kobj_uevent_env, sk_buff, neighbour, netlink_ext_ack, wiphy, wireless_dev, cfg80211_wowlan_wakeup cfg80211_ft_event_params, cfg80211_pmsr_ftm_result, rpc_pipe_msg, geneve_opt, tcp_sock, probe_resp, cfg80211_mgmt_tx_params, ieee80211_mgd_assoc_data rchan_buf, sg_request, fb_cmap, usb_device, urb, usblp, drm_crtc, drm_plane, drm_master, console_font