연구실적_3

2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

# POLaR: Per-allocation Object Layout Randomization

Jonghwan Kim, Daehee Jang, Yunjong Jeong, Brent Byunghoon Kang
KAIST
{zzoru, daehee87, yunjong, brentkang}@kaist.ac.kr

***Abstract*—Object Layout Randomization (OLR) is a memory randomization approach that makes unpredictable in-object memory layout by shuffling and relocating each member fields of the object. This defense approach has significant security effect for mitigating various types of memory error attacks. However, the current state-of-the-art enforces OLR while *compile time*. It makes diversified object layout for each binary, but the layout remains equal across the execution. This approach can be effective in case the program binary is hidden from attackers. However, there are several limitations: (i) the security efficacy is built with the premise that the binary is safely undisclosed from adversaries, (ii) the randomized object layout is identical across multiple executions, and (iii) the programmer should manually specify which objects should be affected by OLR. In this paper, we introduce *Per-allocation Object Layout Randomization* (POLaR): the first *dynamic approach of OLR* suited for public binaries. The randomization mechanism of POLaR is applied at *runtime*, and the randomization makes unique object layout even for the same type of instances. As a result, POLaR achieves two previously unmet security primitives. (i) The randomization does not break upon the exposure of the binary. (ii) Repeating the same attack does not result in deterministic behavior. In addition, we also implemented the TaintClass framework based on DFSan project to optimize/automate the target object selection process. To show the efficacy of POLaR, we use several public open-source software and SPEC2006 benchmark suites.**

## I. Introduction

An *object* is a set of relevant data that often includes security-relevant members, such as pointers. Memory corruption vulnerabilities are majorly caused due to heap errors which involve using objects. This makes the in-object layout of member variables important for the attacker to launch the exploits successfully. For instance, to abuse an object type confusion vulnerability, attackers must calculate the relative position of confused member variables. Similarly, while overwriting an object via buffer overflow, attackers must determine which member variables will be overwritten and which ones will remain intact (e.g., attackers should avoid corrupting irrelevant member variables, which could result in an unexploitable crash). Such examples can be found in various exploit-kits[1].

The adoption of randomness in object layout has been discussed previously to address the problems as mentioned above [39], [43]. In particular, the latest Linux kernel has introduced a new feature that randomizes the structure layout of kernel objects at compile time [16]. However, the technique is designed with an assumption that the binary is hidden from the adversary. Therefore, the current approach of object layout randomization (OLR) breaks if the attacker has *access to the binary*. This assumption is plausible in server-side system environments. The server-side application typically provides its service via controlled network access thus preventing the potentially malicious users from accessing the binary content. However, the assumption breaks for client-side applications as the binary is publicly shared among users.

To apply OLR for publicly disclosed binaries with improved security primitives, we propose Per-allocation Object Layout Randomization (POLaR), which applies an independently randomized memory layout for each object allocation. Because the randomization of POLaR is applied dynamically at *allocation time*, attackers cannot predict the memory layout of an object even if they have access to the binary. Furthermore, POLaR randomizes the memory layout differently for *object instances of the same types*. Therefore, memory corruption attacks such as type-confusion become even more difficult to exploit.

The prototype of POLaR uses LLVM-based compiler techniques for code instrumentation. POLaR analyzes allocation sites of the newly allocated object and automatically finds (i) its initialization code, and (ii) code that accesses the object at a later point, (iii) its de-allocation site, and (iv) its memory copy site. The codes regarding any object randomized by POLaR is instrumented to additionally perform an *object layout lookup* process upon any access to member variables.

As one can imagine, POLaR requires substantial performance cost due to increased memory access for using objects. Therefore, applying POLaR to entire objects could be infeasible in practice. For practicality, we discuss issues regarding automatic object selection which is a best-effort approach for POLaR performance optimization. Whereas existing OLR chooses randomization target objects by programer's manual decision (possibly for any objects), POLaR introduces the *TaintClass* framework to systematically decide the candidate objects for randomization as selective decision process should be mandatory considering the performance impact. The TaintClass framework uses a memory tainting technique based on an open-source data analysis tool (Data Flow Sanitizer) to track objects that are potentially tainted by untrusted input.

The contributions of this paper are summarized as follows.

- To the best of our knowledge, this is the first paper

[1]For example, GreenFlash Sundown, RIG, and Magnitude exploit kits [5] use an Adobe Flash vulnerability, CVE-2018-4878 [2] (use-after-free bug on `DRM` object and assume fixed object layout of `MeM_Arr` class)

978-1-7281-0057-9/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN.2019.00058

IEEE computer society

that introduces the design and implementation of per-allocation object layout randomization *dynamically at runtime*.

- We design and implement Per-object Randomization framework that works with large-scale software such as ChakraCore.
- We design and implement TaintClass framework that automates the object categorization process systematically. Based on the data flow analysis results, TaintClass automatically generates a list of objects that possibly be abused in case of memory corruption; thus require POLaR protection.
- We analyze the existing object layout randomization approaches and discuss its security efficacies and limitations.
- We conduct a series of performance evaluations to investigate the performance cost and applicability of POLaR by applying it to common applications, such as ChakraCore, and SPEC2006.
- We present security case studies based on real-world CVEs to verify the efficacy of per-allocation object layout randomization.

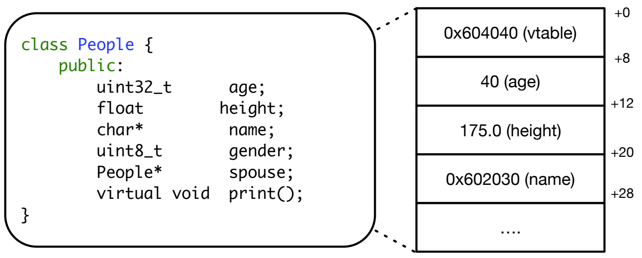
The remainder of this paper is organized as follows. Section II provides the background of the general object structure and memory exploit attacks abusing objects. Section III shows the security effectiveness of the existing OLR and its limitation. Section IV explains the design and implementation of POLaR. Section V evaluates the compatibility and performance of POLaR using various applications and shows the efficacy of TaintClass using real-world CVEs. Section VI discusses metadata safety, optimization, and limitations. Section VII reviews various related studies on object layout randomization. Finally, Section VIII concludes the paper.

## II. Background and Assumption

### A. Attack Model

In this paper, we assume the adversary is capable of providing maliciously crafted input against software that has memory corruption vulnerabilities. Specifically, the paper focus on defending against *heap* memory corruption by introducing *on-the-fly object layout randomization* approach, which has not been explored previously. If the attacker successfully bypasses POLaR (and other mitigations), she can ultimately achieve malicious capabilities such as arbitrary memory read (information leakage), or arbitrary code execution. We do not assume that the attacker has such capabilities before bypassing our defense.

The adversary model in this paper mainly abuses heap corruptions. Many vulnerabilities fall into this category. Use-after-free allows attackers to replace the content of an object with an arbitrary chunk of data. To abuse this error, attackers need to predict the layout of the object; thus, he/she can trigger the intended behavior (i.e., hijacking a specific pointer member variable). A type confusion vulnerability allows an attacker to change the program's memory layout interpretation



```
class People {
    public:
        uint32_t     age;
        float       height;
        char*        name;
        uint8_t      gender;
        People*      spouse;
        virtual void print();
}
```

Fig. 1: Typical memory layout of an object aligned at predictable offset. Most of the objects have at least one pointer member variable (vtable pointer) which makes them important in security perspective.

between two different objects. Attackers abuse this confusion vulnerability by maliciously switching the member variables between two objects. To achieve this goal, it is essential for the attacker to predict the exact memory layout of the objects.

### B. Object Layout

Current state-of-the-art compilers (gcc, clang, and so forth) assume that the data structure memory layout of any objects is deterministic. As shown in Figure 1, the memory representation of code is always deterministic once the compiler decides the relative order and offset of the object member variables. Once the source code is compiled, repeated allocation of the same object type yields the same memory layout. Because the memory layout of the objects is fixed, it is possible to address the object member variables by adding fixed constant to the starting memory address (we refer to it as the base address) of the object; this is an efficient approach. For example, once the code knows the base address of the `People` object in Figure 1, adding a fixed constant of `12` gives the memory location of the `height` member variable. The fixed constant (`12` in this case) of any member variables do not have to be calculated dynamically. Due to the deterministic offsets of the internal object layout, an attacker often analyzes the layout structure of various objects used by the program and uses the information while mounting out-of-bound memory access. Hence, POLaR aims the goal to randomize the internal layout of any objects of the same type.

### C. Linux Kernel's Object Layout Randomization

The current approach of object layout randomization introduced in the recent Linux kernel (`randstruct`) considers randomizing the structure layout *statically at compile time*. The objective of this approach is to divert attackers while they launch their exploits targeted at useful types of fields, such as function pointers or other sensitive data structures, including security credentials (e.g., process uid) involved with a privilege-escalation attempt. This current approach is not only limited to the Linux kernel but also equally considered in general software fortification process. Indeed, previous academic papers discussed this issue [39], [43] as a feature for randomizing the object structure layout at compilation time.

According to the `randstruct`, the following objects are statically randomized during source code compilation.

- Objects declared with `__randomize_layout` annotation tags.
- Objects composed only with function pointers.

In addition, there is `__no_randomize_layout` annotation tag for exceptional usage. The selected object's member variable layout is fully randomized or partially randomized considering the cache line.

### D. DataFlowSanitizer and Fuzzing

DataFlowSanitizer (DFSan) [4] is a tool that allows tracking of the dynamic flow of particular data. To optimize the performance of per-object layout randomization, we implement a TaintClass framework that distinguishes objects into several categories on the basis of their influence by input data (the details are discussed in Section IV). Using the DFSan, TaintClass evaluates whether an object should be randomized We combine a fuzzing technique with DFSan to maximize the data flow coverage. To trace the data flow across the library function calls (such as `memcpy`), DFSan provides a customized ABI list. Fuzzing is a general technique for automatically finding software bugs. However, in POLaR, we use fuzzing to automatically discover mutable objects whose content and allocation/deallocation are affected by a program user who is a potential attacker. To combine the fuzzing process with DFSan, POLaR uses libFuzzer [13] for implementing libpng, libjpeg-turbo's fuzzer.

## III. Security Effectiveness

In this section, we discuss the security efficacy and limitations of the previously introduced object layout randomization (OLR) and highlight the improvement which can be achieved by POLaR.

### A. Revisiting OLR

OLR affects various types of heap memory vulnerabilities. Here, we use type confusion and use-after-free vulnerabilities as representative example cases for discussing the efficacy/limitation of current OLR approach.

*1) OLR and Type Confusion:* Type confusion vulnerability is one of the common memory corruption vulnerabilities found in modern software. The vulnerability is caused by the misinterpretation of an object. For example, the vulnerability allows arbitrary control-flow hijacking in the following scenario. There are two objects: (i) an object `A` that has a function pointer as the `third` member variable; and (ii) another object `B` that has an integer variable as the `third` member variable (assuming that all member variables are equally aligned to 32 bits or 64 bits). Assume that the integer variable of object `B` is fully controlled by the program user (i.e., the variable is the integer ID of the user). In this situation, if the program confuses the type between objects `A` and `B`, the user can hijack the function pointer of object `A` because the program will interpret the user-controllable integer value of object `B` as a function pointer member of object `A`. The exploitability of type confusion vulnerabilities is significantly dependent on the in-object memory layout.

Therefore, OLR is effective against type confusion vulnerabilities because it hides the in-memory layout of member variables from attackers. However, triggering the type confusion errors yields deterministic result under OLR because the two confused member variables between two different objects will always be the same although the attackers do not know what such members are. The attacker can repeat triggering the same result of type confusion. The advantage of POLaR in this aspect is that it has a significant impact on nurturing object type-confusion vulnerabilities by removing memory layout determinism even for member variables of identical object types.

*2) OLR and Use-After-Free:* Data objects are dynamically allocated to the heap as required and should be freed when they are no longer required. However, a program could accidentally reference a freed object pointer (dangling pointer) and use the object as if it were still allocated, which is known as use-after-free. The use of dangling pointers itself does not cause any harm (except that free metadata overwrites the object) to the program. However, if the application recycles the freed object for the attacker-controlled data allocation, it becomes a severe security breach, allowing the attacker to manipulate the entire content of an object. To abuse use-after-free vulnerabilities, attackers must successfully re-allocate the dangling pointed memory space and place the maliciously crafted fake object. The key here is that the attacker expects how the program will interpret the fake object on the basis of static analysis of the object memory layout. Therefore, if the object memory layout is dynamically determined at runtime, the attacker's exploit code cannot define the fake object as intended. Consequently, as with the type confusion vulnerability, the randomness of OLR also hinders the exploitation of use-after-free vulnerabilities. Unfortunately similar to the type-confusion example discussion, OLR allows deterministic reproduction of bug triggering.

### B. Improving the security of OLR

Existing OLR approach deludes attackers from abusing broad range heap vulnerabilities such as use-after-free and type-confusion. However, there are some limitations in the current version of OLR. First of all, OLR is only effective in case the affected binary is hidden from the adversary. Second, the security effectiveness of OLR is less effective against repeated execution attempts as the changed object layout remains equal across executions. POLaR addresses such limitations by adopting the OLR dynamically at *allocation time*. We note that POLaR changes the in-object memory layout (optionally adding unused member variables to increase the entropy) even for objects that are of the same type. Figure 2 depicts how POLaR changes the memory layout. This increases the security effectiveness for type-confusion attacks, repeated exploitation attempts, and so forth. In particular, POLaR improves OLR in terms of the following problems.

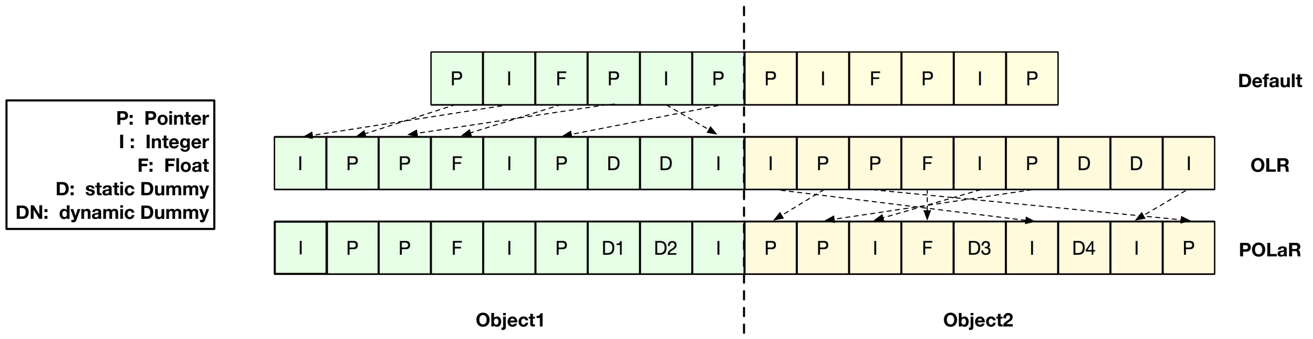


Fig. 2: The effectiveness of POLaR over OLR. Per-allocation randomization changes the in-object memory layout for object instances of the same type. OLR does not provide such diversity as the randomization is statically applied based on source code.

*1) Hidden Binary Problem:* The major limitation of OLR is that it randomizes the object layout at *compile time*. The randomization methods, randomized structure information is stored inside the binary. Reverse engineering the binary would trivially reveal such information. This usage model confines the security efficacy of OLR only to the server-side applications which do not provide the application binary to the users. Consequently, the OLR approach cannot be applied for client-side binaries which are publicly shared among multiple users. POLaR address this problem by enforcing the randomization dynamically at allocation time. Because the object memory layout is randomized at runtime, POLaR maintains its security effectiveness regardless of the availability of the binary.

*2) Reproduction Problem:* Another limitation of current OLR approach is that the randomized object layout is unchanged across execution. This makes determinism for triggering an object corruption vulnerability such as type confusion, use-after-free, and use-before-init. Because the randomization happens upon compilation phase, any runtime object instance that shares the same type shares the same layout. Also, the once-randomized object layout remains same across multiple executions. Therefore attacker can observe deterministic behavior by triggering the memory corruption with the same input data. This allows the attacker to infer and analyze the changed object layout. POLaR address this problem by (i) randomizing the layout at runtime, and (ii) randomizing the layout even for the same object types. We explain the design and implementation details in Section IV to show how such dynamic randomization is possible.

*3) Object Selection Problem:* Besides the security impact, another limitation of OLR is that there is no systematic way of choosing objects that require randomization[2]. To maximize the performance efficacy, the target object selection policy of OLR is important. At this point, the selection is manually performed by programmers. Thus far, there is no standard, and there has been no academic discussion regarding which objects should be protected by OLR.

[2]Applying randomization against all objects is discouraged owing to performance and other reasons (e.g., network packet protocols).

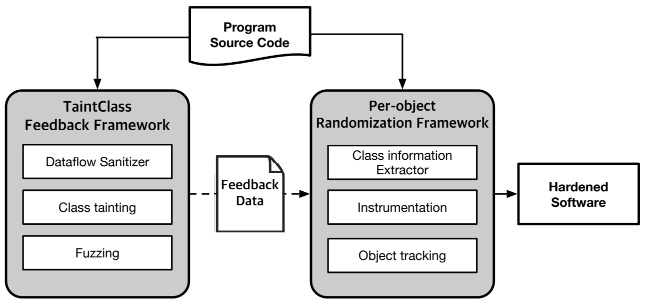
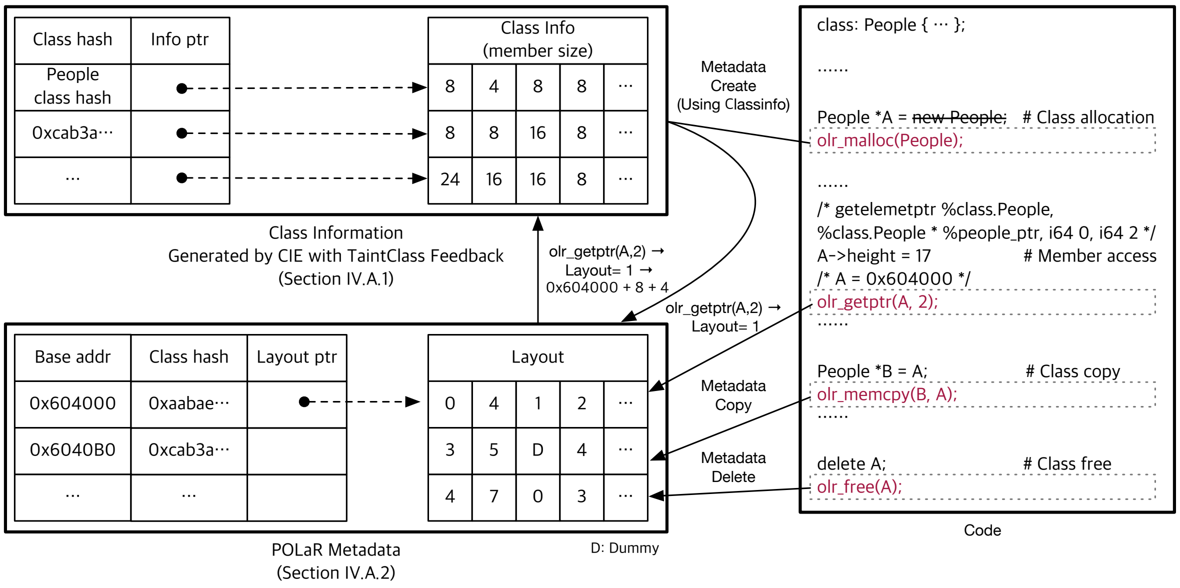


Fig. 3: Overview of POLaR. The TaintClass framework provides the object list as feedback information for the randomization module. TaintClass is *NOT* mandatory for executing POLaR hardened binary.

Heuristic selection can easily miss some important objects as protection targets. For example, in the case of CVE-2018-5703 [3] (use-after-free vulnerability of a Linux kernel object), the vulnerability could have been mitigated if the OLR was properly applied against the `sock` object in `net/sock.h`. Unfortunately, OLR (via `randstruct`) in the latest Linux kernel (v4.20-rc3) did not annotate this object for randomization thus allowed attackers to abuse the use-after-free. We discuss further details of the object selection issue while introducing our TaintClass framework, which is a sub-component of POLaR.

## IV. Design and Implementation

In this section, we demonstrate the design of POLaR and explain how our system can enhance the security level of previous OLR. Figure 3 shows the overall architecture of POLaR. There are two frameworks consists POLaR: (i) the Per-object randomization framework which enables per-allocation randomization based on LLVM/Clang and (ii) the TaintClass feedback framework which tracks and analyzes the user input data for POLaR's target object selection based on DFsan. Overall, POLaR is designed on top of LLVM and DFSan for its back-end support.


Class Info
(member size)
Metadata Create (Using Classinfo)
Class Information
Generated by CIE with TaintClass Feedback
(Section IV.A.1)
olr_getptr(A,2) → Layout= 1 → 0x604000 + 8 + 4
olr_getptr(A,2) → Layout= 1
Layout
Metadata Copy
Metadata Delete
POLaR Metadata
(Section IV.A.2)
D: Dummy
Code

| Class hash | Info ptr |
|---|---|
| People class hash | ● |
| 0xcab3a··· | ● |
| ··· | ● |

| | | | | |
|---|---|---|---|---|
| 8 | 4 | 8 | 8 | ··· |
| 8 | 8 | 16 | 8 | ··· |
| 24 | 16 | 16 | 8 | ··· |

| Base addr | Class hash | Layout ptr |
|---|---|---|
| 0x604000 | 0xaabae··· | ● |
| 0x6040B0 | 0xcab3a··· | |
| ··· | ··· | |

| | | | | |
|---|---|---|---|---|
| 0 | 4 | 1 | 2 | ··· |
| 3 | 5 | D | 4 | ··· |
| 4 | 7 | 0 | 3 | ··· |

```
class: People { ··· };

......

People *A = ~~new People;~~   # Class allocation
olr_malloc(People);

......
/* getelemetptr %class.People,
%class.People * %people_ptr, i64 0, i64 2 */
A->height = 17              # Member access
/* A = 0x604000 */
olr_getptr(A, 2);
......

People *B = A;               # Class copy
olr_memcpy(B, A);
......

delete A;                    # Class free
olr_free(A);
```

Fig. 4: Implementation details of POLaR. To support per-allocation randomization, POLaR requires binary instrumentation based on LLVM IRs.

### A. Per-object Randomization Framework

*1) Class Information Extractor:* The Per-object Randomization framework requires the source code of the program to apply LLVM instrumentation later. The Class Information Extractor (CIE) module takes the program source code and TaintClass feedback data as input and generates information for the target architecture regarding the structure and class declaration using LLVM API. The information generated by the CIE module includes class size, member types, and member size. This information is embedded in executable files, and it is passed to the POLaR runtime library; it will be used later for the randomization.

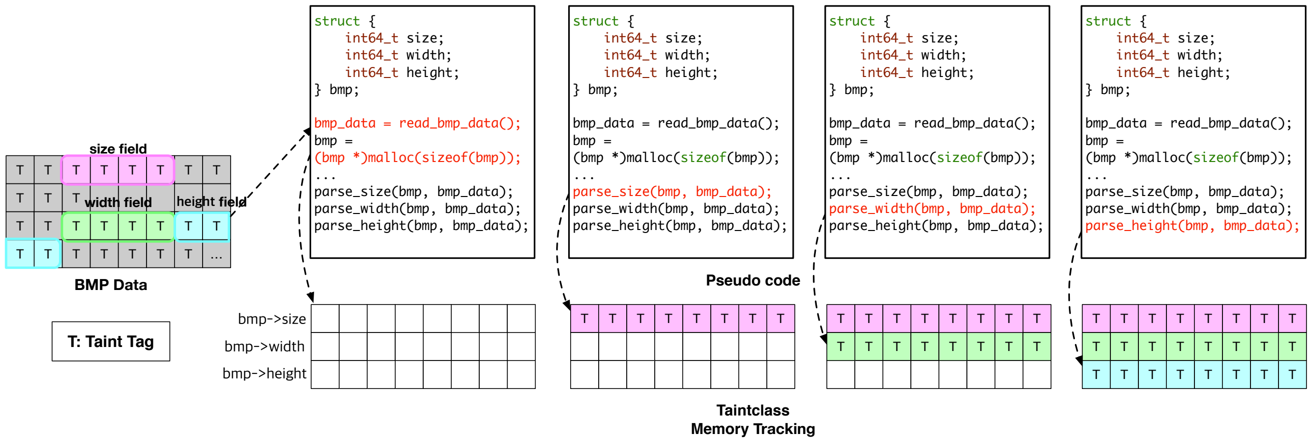
*2) Instrumentation:* Randomizing the memory layout of the same object type involves various implementation challenges. The first challenge is instrumenting the existing code to interpret an object layout before using it. This interpretation process is automatically applied to the generated binary using LLVM without changing the original source code. As the LLVM-pass automatically instruments the code accessing the objects, programmers do not have to consider the permuted memory layout of the object. As a result, the programmer does not have to alter any syntax. In particular, the following functions and LLVM IRs are instrumented by POLaR: (i) allocation/deallocation types (malloc, alloca, free, etc.) of functions, (ii) `getelementptr-like` LLVM instructions (getelementptr, extractvalue, insertvalue), and (iii) memcpy types (memcpy, memmove, etc.) of functions.

The allocation functions need to be changed to generate the object layout information of each allocated object and maintain the information. Based on the information provided by LLVM IR regarding the object type upon allocation, POLaR instruments the allocation functions to use the object information for randomizing the object layout generated by CIE and save the allocated memory region and randomized layout as metadata into POLaR object tracking library. Deallocation functions are also instrumented to remove the metadata generated during the previous allocation.

The getelementptr-like instructions [6] are typical LLVM instructions for retrieving the address of member variables in a class. For example, when program accesses the memory of the `height` variable of the `People` class in Figure 1, LLVM translate this function to `getelemetptr %class.People, %class.People *%people_ptr, i64 0, i64 2`. (where 0 is the indexing for the array type and 2 is the position of the height variable. vtable: 0, age: 1, height: 2). We instrument these getelementptr-like instructions, modifying the way to access the member variable. Figure 4 depicts the details how the POLaR instrumented objects are accessed with metadata.

Memory copying APIs, such as `memcpy`, are also instrumented for POLaR. For example, if the POLaR-applied object is copied into another memory region, a duplicate copy of the object can be created without any heap allocation attempt. This duplicate copy is also subject to POLaR randomization. By instrumenting the memory-copy APIs, POLaR randomizes the object layout for the duplicate copy and additionally creates valid mapping information for accessing the copy. This feature could be disabled with the configuration for performance-purposes, but the current implementation considers this feature enabled by default.

*3) Object Tracking:* The second challenge for POLaR is tracking the object allocation/deallocation, member access, and memory copy sites. As described in Section IV-A2, we instrumented these instructions and changed the execution flow



```
struct {
    int64_t size;
    int64_t width;
    int64_t height;
} bmp;

bmp_data = read_bmp_data();
bmp =
(bmp *)malloc(sizeof(bmp));
...
parse_size(bmp, bmp_data);
parse_width(bmp, bmp_data);
parse_height(bmp, bmp_data);
```

```
struct {
    int64_t size;
    int64_t width;
    int64_t height;
} bmp;

bmp_data = read_bmp_data();
bmp =
(bmp *)malloc(sizeof(bmp));
...
parse_size(bmp, bmp_data);
parse_width(bmp, bmp_data);
parse_height(bmp, bmp_data);
```

```
struct {
    int64_t size;
    int64_t width;
    int64_t height;
} bmp;

bmp_data = read_bmp_data();
bmp =
(bmp *)malloc(sizeof(bmp));
...
parse_size(bmp, bmp_data);
parse_width(bmp, bmp_data);
parse_height(bmp, bmp_data);
```

```
struct {
    int64_t size;
    int64_t width;
    int64_t height;
} bmp;

bmp_data = read_bmp_data();
bmp =
(bmp *)malloc(sizeof(bmp));
...
parse_size(bmp, bmp_data);
parse_width(bmp, bmp_data);
parse_height(bmp, bmp_data);
```

Fig. 5: Taint propagation example of the TaintClass framework against an untrusted BMP file input.

to be handled by the object tracking library instead. To apply a different memory layout for each object, Object tracking library tracks the object initialization code and instruments it to obtain the metadata and randomize each object. The library also implements the dynamic member offset calculation on the basis of the POLaR metadata stored upon object allocation. The randomization process of POLaR not only permutes the sequence of existing member variables but also inserts dummy member variables to increase the randomness entropy.

Furthermore, inspired by [25], [26], POLaR not only use the dummy variables for increasing the randomization entropy but also utilize it as a booby trap for detecting object corruption. For example, to protect a function pointer (inside an object) against buffer overflow bug, POLaR prepends dummy variables adjacently to the function pointer and utilize it as a booby trap for overflow detection. This mechanism can detect an attack attempt on the object in advance. Additionally, POLaR detects obvious use-after-free attempts while regulating object access using the metadata information.

## *B. TaintClass Framework*

TaintClass is a feedback framework for POLaR target selection. The objective of TaintClass is to avoid heuristic/manual selection (existing approach) of the randomization target objects as well as unnecessary randomization thus optimize the performance of POLaR. The basic algorithm of the TaintClass framework aims to find the input-dependent objects that are controllable by a program user. For instance, some objects only appear inside the program heap temporarily to setup the initial graphic components and thus irrelevant for potentially dangerous input parsing. Applying randomization to such an object is meaningless. Existing OLR approaches manually find the objects that interact with untrusted inputs in order to adopt randomization against them. In POLaR, we automate this process systematically by using the TaintClass feedback framework, which aims to find such objects automatically by using a data-flow tracking technique based on DFSan. We note that TaintClass itself does not implement a data-flow tracking algorithm. It utilizes DFSan's support for labeling data of our interest in byte granularity. Once we configure the initial data source, DFSan internally tracks the data flow dependency based on shadow memory implementation.

*1) Class Tainting:* The attack model in this paper assumes that adversary abuses memory corruption with arbitrary crafted input data. The goal of the attacker is to trigger memory corruption bugs successfully (e.g., type confusion, use-after-free) for exploitation primitive. The majority of client-side applications takes untrusted input data with various interfaces. For example, web-browsers takes JavaScript as an untrusted input for its rendering, and various document parsers take an untrusted input in multiple forms (e.g., files or network stream). The vulnerability is triggered while processing such untrustworthy input data. To process such data, program interacts with various objects inside the heap, which can be potentially abused by attackers.

To automatically enumerate the objects potentially subject to the untrusted input, the TaintClass framework uses DFSan to analyze the data flow and identify the objects affected by the initially given input. Using DFSan, TaintClass taints the in-memory propagation of the input data flow starting from the initial input buffer or memory mapping. In case the input is given as files, the initial buffer/mapping of the input is handled by instrumenting the responsible system calls and APIs (e.g., fread, and MapViewOfFile). While tracking the memory propagation, TaintClass searches for the case in which the propagation affects the content/allocation/deallocation of any objects. If such objects are discovered, TaintClass collects propagation information, which will be used later by POLaR. The execution of TaintClass is orthogonal to the execution of the hardened binary. We note that TaintClass and Per-allocation Randomization framework can work in parallel; feeding each other's data.

Figure 5 shows a simplified example of the TaintClass framework tracking the tainted information from a user-supplied BMP file. In the source code, TaintClass initially maps the file content to memory (e.g., `read_bmp_data()`).

TABLE I: Object list reported by TaintClass framework against SPEC2006 benchmark applications. The life-cycle and contents of the reported objects are potentially affected by an untrusted input.

| App | # of tainted objects | Several samples of tainted objects |
|---|---|---|
| 400.perlbench | 20 | sv, stat, cop, _sublex_info, jmpenv, logop, unop, scan_data_t, RExC_state_t, ... |
| 401.bzip2 | 3 | bzFile, UInt64, spec_fd_t |
| 403.gcc | 33 | realvaluetype, ix86_address, type_hash, stat, cb_args, mem_attrs, addr_const, ix86_args, ... |
| 429.mcf | 2 | network, basket |
| 445.gobmk | 21 | move_data, SGFTree_t, gg_rand_state, worm_data, dragon_data, Hash_data, string_data, ... |
| 456.hmmer | 4 | seqinfo_s, comp, exec, ssifile_s |
| 458.sjeng | 2 | move_s, move_x |
| 462.libquantum | 0 | - |
| 464.h264ref | 17 | InputParameters, decoded_picture_buffer, pic_parameter_set_rbsp_t, ImageParameters, ... |
| 471.omnetpp | 10 | cSimulation, cHead, _Task, TOmnetApp, cPar, cArray, cPar::ExprElem, MACAddress |
| 473.astar | 7 | wayobj, way2obj, regmngobj, workinfot, createwaymnginfot, regboundobj, regobj |
| 483.xalancbmk | 59 | XalanDOMString, XObjectPtr, XalanQNameByValue, XalanQNameByReference, MutableNodeRefList, ... |
| libpng 1.6.34 | 8 | png_struct_def, png_info_def, png_xy, png_XYZ, png_color16_struct, png_text, png_time_struct, ... |
| libjpeg-turbo 1.5.2 | 8 | _tjinstance, bitread_working_state, savable_state, jpeg_component_info, j_decompress_ptr, ... |
| Chakracore 1.10 | 42 | Js::HashedCharacterBuffer, Js::OpLayoutT_Reg1, JsUtil::CharacterBuffer, Js::FunctionBody, ... |

In this step, the memory contents are tainted with tags. Later the initially tagged memory contents are traced when the information is propagated via APIs such as `memcpy` or instructions that reference such memory. In case the tainted information propagates to an object, we consider the object to be potentially affected by the untrusted user input. In addition, TaintClass identifies exactly which object members (pointer type or non-pointer type) are tainted. This information is used later for optimizing the efficacy and dummy variable insertion of POLaR.

*2) Increasing Taint Coverage:* DFSan is an efficient tool for tracking taint propagation. However, if the taint propagation is dependent on the contents of the input data, DFSan itself is insufficient to achieve our objective. To address this issue, we have incorporated the input generation of DFSan with a coverage-guided input generation module from libFuzzer. In general, libFuzzer is a coverage-guided test case generator designed to discover bugs. In the TaintClass framework, we use only the coverage-guiding module and combine its algorithm with the DFSan input case generation. As a result, our TaintClass framework effectively distinguishes objects that are dependent on the untrustworthy input and provides this information to the main randomization module of POLaR. We evaluate the efficacy of the TaintClass framework by using it against open-source software and comparing the results with object lists involved in publicly disclosed vulnerability exploitation processes.

## V. Evaluation

In this section, we evaluate the POLaR from three perspectives: (i) compatibility with existing software, (ii) performance cost, and (iii) efficacy of the TaintClass framework. In the case of compatibility testing, we choose the libpng library, libjpeg-turbo library, V8 Javascript engine of the Chrome web browser, and ChakraCore [1] JavaScript engine of the Edge web browser. To evaluate the performance, we use the SPEC2006 benchmark and various JavaScript benchmark suites. Finally, for TaintClass evaluation, we survey public CVE attack cases against libpng heap vulnerabilities. The evaluation environments are Intel E5-2630 v3 (2.40 GHz) CPU and 128GB RAM, running the 64-bit version of Ubuntu 16.04 Server. For the evaluation, we do not compare the performance of OLR and POLaR as their target applications, and key designs vastly differ from each other considerably. The performance cost of OLR is assumed to be negligible, as it does not impose a significant penalty besides reduced cache utilization.

### A. Compatibility

To demonstrate the compatibility of POLaR with existing applications, we applied POLaR against all SPEC benchmark applications, libjpeg-turbo, libpng, V8, and ChakraCore. First, we applied POLaR to the entire set of objects without using the TaintClass framework. In this case, we discovered some compatibility errors only in V8 JavaScript engine. We found that the reason for such failure is due to the implementation techniques leveraged in V8 custom garbage collector (Orinoco). The current version of POLaR instrumentation does not correctly handle such codes (can be resolved with additional engineering efforts) therefore we excluded V8 at this point. ChakraCore did not suffer such issues as it uses an ordinary mark-and-sweep garbage collector.

Later, we applied TaintClass for target applications and evaluated correctness with benchmarks. Table I lists the objects reported by TaintClass. To maximize the code coverage for the object discovery process, the TainClass Framework uses a LibFuzzer with Edge-level code-coverage instrumentation. This fuzzing step (which is not required for each execution) requires several hours, as the input case generation for DFSan.

From the results in Table I, TaintClass did not mark any objects of SPEC2006's 462.libquantum [17] application for POLaR randomization. The application is a quantum computing simulator that obtains input data via the main function parameter. The input is directly propagated for floating point operations; thus there is no object involved. We have confirmed this manually by analyzing the application source code.

During the evaluation, we also audited the source code and manually profiled the objects that are dependent on the

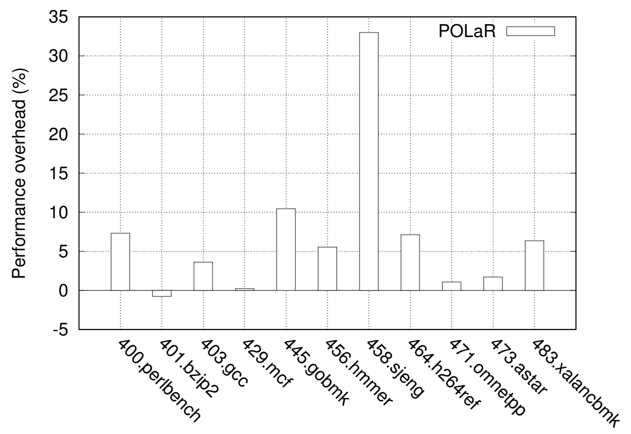


Fig. 6: Performance overhead of POLaR based on the SPEC2006 benchmark.

TABLE II: Performance overhead of POLaR based on the ChakraCore benchmarks.

| Benchmark | Result | | | Ratio |
|---|---|---|---|---|
| | Default | POLaR | DIFF | |
| Sunspider (time) | 2642.6 ms | 2646.9 ms | 16.7 | 0.20% |
| Kraken (time) | 10369.4 ms | 10386.1 ms | 4.3 | 0.20% |
| Octane (score) | 2728.7 | 2699 | -29.1 | -1.10% |
| Jetstream (score) | 117.1 | 117.9 | 0.8 | 0.70% |

* Sunspider, Kraken's Result: smaller is better

untrusted input. Based on the manual discovery and automated result of discovered objects, we evaluate the false-positive ratio of the TaintClass framework. We discuss the false-negative discovery issues in Section VI.

## B. Performance

The performance impact of POLaR is expected to be high due to the significant amount of memory access instrumentation. With POLaR, memory access of randomized object members will require additional steps of pointer dereferencing in order to calculate the actual offset randomized by POLaR. Therefore, the performance impact will be high against applications that excessively access object members, and it will be low for applications that focus on other operations, such as I/O or arithmetics. To optimize the performance cost of additional lookup procedure of member variable offsets, POLaR implements the hashtable-based caching mechanism that store the previous result of the lookup procedure. Furthermore, Polar remove the duplicate metadata when two objects have the same randomized memory layout.

Figure 6 summarizes the SPEC2006 performance evaluation. The performance overhead of the SPEC2006 benchmark is around 5%, except for the sjeng [18] program. Sjeng is a chess engine that takes the initial state of the chess pieces as the only program input. With the given input, sjeng creates objects for each state of chess pieces which are subject to POLaR randomization. Due to the characteristic of the sjeng program, the major bottleneck of the program's performance is object allocation/deallocation, which constitutes the worst performance evaluation case. Table III lists the number of allocation/free, member variable access, and cache hit against the randomized objects.

POLaR was also applied to ChakraCore v1.10, a JavaScript rendering engine of the Microsoft Edge web browser. We used the standard test cases[3] and JavaScript benchmarks (Kraken [12], Octane [15], Sunspider [19], Jetstream [11]) provided by ChakraCore for the official testing purpose. All the test cases worked correctly in both cases, and Table II summarizes the benchmark results.

From the ChakraCore benchmark results, we can observe approximately 1% performance slowdown in Sunspider, Kraken, and Octane. In the case of Jetstream, no measurable performance variation was observed. We suspect that the reason for such low-performance cost is due to the performance optimization in ChakraCore engine minimizing the heap allocation/deallocation operations.

## C. Correctness of TaintClass Framework

The objective of the TaintClass framework is to automatically discover objects whose contents and life-cycles are affected by untrusted user inputs and thus provide information to the POLaR randomization module. To evaluate the correctness of the TaintClass framework, we manually inspect the application source codes and exploit codes to enumerate all the objects that are abused by attackers in their exploit. Later, we compare the results of our analysis against the automatically generated object list of TaintClass. For the evaluation, we analyzed 35 CVE-based attacks against libpng and found heap vulnerability cases that are suited for the evaluation.

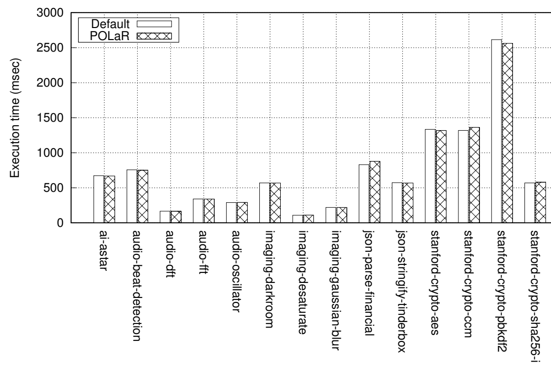
The objects discovered by the TaintClass framework are based on 3 hours (including fuzzing step) of the discovery process. Table IV summarizes the evaluation. The TaintClass framework successfully included all the objects that we discovered by manually analyzing the exploitation while excluding irrelevant objects in terms of the exploitation. We note that TaintClass object selection result is a best-effort approach based on tainting technique. The list is not theoretically proven to be the unique correct result.

## VI. Discussion

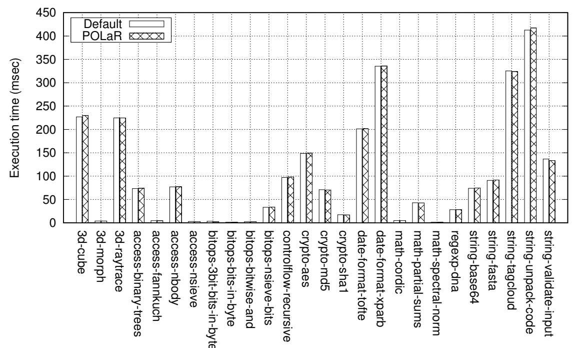
## A. Metadata Safety

POLaR keeps the randomized offset information per each object as its metadata. There are some chances in which vulnerabilities bypass our POLaR protection (e.g., via logical bugs) and corrupt the metadata information of POLaR. At this point, POLaR does not provide a solution for securely keeping its metadata secret against other than hiding its memory location. However, we plan to adopt modern state-of-the-art solutions to protect the metadata against information leakage attacks. For example, recent defenses such as Intel MPX [7], SGX [9], MPK [8], and ARM TrustZone [10] provide special memory regions for protecting critical information against unintended memory disclosure.

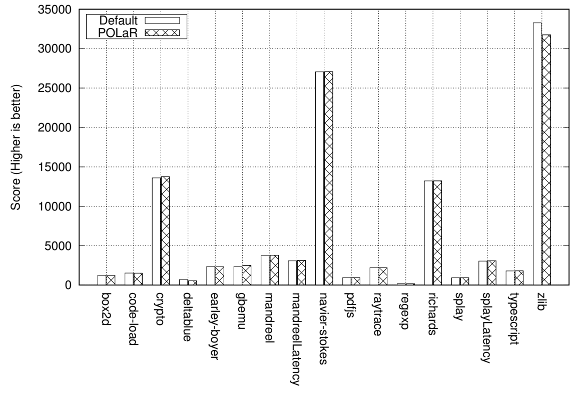
[3]ChakraCore provides standard test codes for its benchmark.



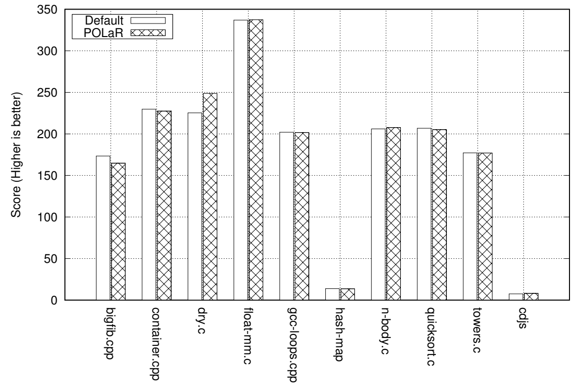
(a) Kraken



(b) Sunspider



(c) Octane



(d) Jetstream

Fig. 7: Perfomance evaluation of POLAR against Chakracore benchmark

## B. Instrumentation Problems

POLaR uses LLVM techniques to automatically instrument code regarding object access. The instrumentation is automatically handled in most cases by using the source code syntax information. However, there is some exceptional code that does not use common syntaxes for accessing objects. For example, instead of referencing object member variables via their declared names, some code manually calculates the offset of a member variable from the starting address of an object. In addition, complicated garbage collection codes in Chrome V8 are yet incompatible with POLaR technique. However, POLaR covers most of the general implementation issues including aliasing/casting and sub-typing techniques. We also note that regardless of the complicated implementation problems, serializable objects and packets are often unsuited for instrumentation due to protocol conflicts.

## C. Tainting Issues

The TaintClass framework for POLaR is an approach based on DFSan by tracking the dependency between objects and attacker-controllable input data. We combine the DFSan and a coarse-grained fuzzing technique with LibFuzzer to implement an automated object discovery process for the TaintClass framework. However, DFSan lacks tainting support for some of the API calls. Nevertheless, this is an implementation issue which can be handled with additional engineering effort. There is no fundamental design problem with this issue. TaintClass has already implemented additional supports for taint propagation which DFSan lacks in its implementation. Although the taint propagation of the TaintClass framework is properly implemented, there is no guarantee of discovering all the objects that are potentially relevant to the exploit. There could be certain cases where the TaintClass framework excludes an object from randomization although untrusted inputs can ultimately control it.

## VII. Related Work

## A. Existing Object Layout Randomization

The basic concept of object layout randomization has been discussed previously in the literature. For example, Data Structure Layout Randomization (DSLR) [39] is a closely related previous study on object layout manipulation. The paper introduces the concept of *object layout randomization* and demonstrates its efficacy by implementing it for kernels. DSLR randomizes the relative order of member variables

TABLE III: The number of allocation/free, member variable access, and cache hit attempts against the randomized objects.

| App | # of | | | | |
|---|---|---|---|---|---|
| | Allocation | Free | Memcpy | Member access | Cache hit |
| 400.perlbench | 5,645K | 0 | 3312 | 80B | 74B |
| 401.bzip2 | 36 | 37 | 0 | 34M | 28M |
| 403.gcc | 51M | 50M | 0 | 0 | 0 |
| 429.mcf | 1 | 0 | 0 | 9,105K | 9,105K |
| 445.gobmk | 4000 | 0 | 0 | 72B | 65B |
| 456.hmmer | 1 | 1 | 0 | 4,291K | 3,677K |
| 458.sjeng | 20M | 20M | 18M | 151B | 128B |
| 464.h264ref | 450 | 450 | 298M | 1,993M | 1,871M |
| 471.omnetpp | 132 | 1 | 1 | 803 | 406 |
| 473.astar | 12 | 12 | 354K | 204 | 144 |
| 483.xalancbmk | 28,686 | 19,985 | 26 | 1,000K | 696K |

TABLE IV: Automatically discovered `libpng` objects that were used for exploitation.

| CVE | Descriptions | Exploit-related Objects |
|---|---|---|
| 2016-10087 | null pointer dereference | png_{info, struct}_def |
| 2015-8126 | heap overflow | png_{info, struct}_def png_color |
| 2015-7981 | out of bounds read | png_{struct_def, time_struct} |
| 2015-0973 | heap overflow | png_{struct_def, byte} |
| 2013-7353 | integer overflow | png_{struct, info}_def png_unknown_chunk |
| 2011-3048 | heap overflow | png_{struct,info}_def png_text |

inside kernel objects. The primary objective of this approach is to hinder the data structure manipulation attempts of rootkits. In addition, dummy member variables are inserted while randomizing the object layout in case the number of existing member variables is insufficient. The basic idea of object layout randomization is introduced in DSLR; however, its approach is static. The major advancement of POLaR over DSLR is that whereas the previous approach adopts the randomization at *compile time*, POLaR applies randomization at *allocation time*. This leads to a significant difference in terms of security efficacy, implementation techniques, and performance. One of the main advantages of POLaR that DSLR lacks is the randomization of the object layout among the *same type of objects*. Using LLVM-based instrumentation, POLaR diversifies the object layout of multiple object instances of the same type.

Record Field Order Randomization [43] (RFOR) also introduced the concept of object layout randomization previously. As with DSLR, the randomization phase of RFOR is at *compilation time*. This changes the object layout *per-binary*. While the details differ between DSLR and RFOR, both are based on the same main idea for randomizing the sequence of in-object member variables, which is statically assuming that the binary is not revealed to attackers. We highlight the major drawbacks of these previous studies, i.e., the security effectiveness is easily broken once the attacker analyzes the binary. The threat model and assumption of POLaR are based on the attackers with greater capabilities (attackers have complete access to the binary). Such an assumption advances the state of the art while imposing various new challenges regarding design and implementation which we addressed in this paper.

## B. Diversifying Various Software Components

Numerous studies have introduced the idea of various randomization approaches for exploit defense. Some approaches [44] [31] [36] randomizes the overall memory layout. Others [24], [30], [34], [27], [35], [33] changes the instructions to randomize various semantics as a defense measure. Since Android 7.0 randomizes the shared library loading sequence [14].

The majority of these previous studies aim to hinder attackers from predicting the general memory layout or make the order and relative distances *between objects* less predictable. The concept of shuffling and randomizing the order/distance of heap data may seem similar to object layout randomization. However, the details and its ramifications differ significantly. The approach of randomizing the inter-object layout for each allocation can be implemented without instrumenting the code regarding object access, and it has orthogonal effects regarding exploit mitigation. Unlike the approaches for overall heap layout randomization (inter-chunk randomization), we focus on randomizing the in-object layout of member variables. To support such in-object layout randomization, the code referencing each member variables of the object has to be instrumented. Due to the code instrumentation, POLaR requires a higher performance cost compared to inter-chunk randomization. However, POLaR addresses memory attacks (such as type confusion), which could not be mitigated by inter-chunk randomization approaches.

## C. Other Exploit Mitigations

Many studies aim to protect memory object bounds and dangling pointers deterministically. They typically enforce the bounds by inserting checks using compile-time instrumentation. There are two primary approaches for deterministic bounds protection. The redzone-based approach [41], [32] is the most popular memory safety technique used by large-scale software projects and fuzzers. It works by checking the access permission of the given address before any memory access. To detect overflows, inaccessible regions called `redzones` are inserted between objects. Freed objects are marked inaccessible, and the address reuse is delayed via quarantine. Its simple design minimizes the performance impact and achieves good compatibility. However, redzone-based approaches allow out-of-bound access that falls into other objects. Pointer-based approaches [40], [20], [47], [37], [28], [29] tracks the memory access capabilities of pointers. Each study differs in terms of how the capabilities are managed and when the checks are performed. Pointer-based approaches can prevent more out-of-bound accesses than redzone-based approaches because access permission is dependent on pointers, not addresses. However, most studies on these approaches focus on protecting bounds between chunks. Checking in-object overflows has not been fully explored thus far because it may cause a high-performance overhead, or raise compatibility concerns with existing software. POLaR deals with memory corruption exploits, including in-object overflows.

In addition to memory safety instrumentations, a number of studies on regarding dangling pointers mitigate use-after-free vulnerabilities by either invalidating or detecting the use of such pointers. DangNull and FreeSentry [38], [46] keep map information regarding objects and their pointers. Upon freeing an object, all the related pointers are set to an invalid memory address. Dangling pointer access would then cause a segmentation fault. DangSan [45] optimizes this approach to scale heavily multithreaded applications with many allocations. Undangle [22] detects dangling pointers at an early stage where they are created. This approach uses taint analysis to track pointers originating from the same memory object. At any point, the origin of a given pointer can be determined. By checking if the object is freed, it is possible to detect the creation of dangling pointers. While these solutions are effective at preventing dangling pointer usage, corruptions by type confusion vulnerabilities are not covered. POLaR mitigates both type confusion, overflow, and use-after-free attacks.

Another type of memory corruption defense focuses on securing vtables. For example, CFIXX [21] mitigates C++ vtable hijacking exploits. This approach is somewhat similar to shadow stack approaches [23], [42]. When a C++ object is created, its vtable pointer (e.g., commonly denoted as `vptr`) is initialized by the appropriate constructor. CFIXX stores a shadow copy of this `vptr` inside a safe region. Then, when a virtual function call is needed, the `vptr` in the object and its shadow copy are compared. Any modification of the `vptr` can be detected in this way. However, corruptions of other member variables are not protected. Function pointers or security critical values inside the object are still subject to malicious manipulation. POLaR provides mitigation against malicious access to any type of member variable.

## VIII. Conclusion

In this paper, we introduced POLaR, a first dynamic approach of OLR, and addressed various limitations of current OLR approach with other improvements. Because POLaR randomizes the in-object memory layout dynamically at *allocation time*, the in-object memory layout between any object instances (including those of the same type) becomes unpredictable. The major improvement of POLaR over OLR can be summarized as follows: (i) Dynamic randomization approach of POLaR allows public sharing of the fortified binary, (ii) randomization support for identical object type prevents deterministic reproduction of bug triggering, and (iii) TaintClass framework (which is a sub-component of POLaR) provides automatic selection of object targets based on data-flow analysis; improving the previous manual/heuristic selection approach of OLR. Design and implementation of POLaR is based on LLVM instrumentation. Performance and compatibility are evaluated with open-source software and benchmarks including libpng, libjpeg-turbo, ChakraCore, and SPEC2006. According to the evaluation, the performance cost of POLaR is around 5% in the SPEC2006 benchmark and around 1% in the JavaScript application benchmarks.

## Acknowledgment

We thank the anonymous reviewers for their constructive feedback. We also thank Long Wang, Timothy Tsai for their helpful feedback. This research is supported by the NRF(NRF-2017R1A2B3006360), IITP(IITP-2017-0-01889) and ONR(N00014-18-1-2661) grants.

## References

[1] "Chakracore is the core part of the chakra javascript engine that powers microsoft edge," https://github.com/Microsoft/ChakraCore, Online; accessed 29-March-2019.
[2] "Cve-2018-4878," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4878, Online; accessed 29-March-2019.
[3] "Cve-2018-5703," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5703, Online; accessed 29-March-2019.
[4] "Dataflowsanitizer," https://clang.llvm.org/docs/DataFlowSanitizer.html, Online; accessed 29-March-2019.
[5] "Exploit kits: Spring 2018 review," https://blog.malwarebytes.com/cybercrime/2018/06/exploit-kits-spring-2018-review, Online; accessed 29-March-2019.
[6] "getelementptr instruction," https://llvm.org/docs/LangRef.html#getelementptr-instruction, Online; accessed 29-March-2019.
[7] "Intel corporation. 2013. introduction to intel(r) memory protection extensions," https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, Online; accessed 29-March-2019.
[8] "Intel corporation. intel 64 and ia-32 architectures software developer's manual. april 2016." Online; accessed 29-March-2019.
[9] "Intel® software guard extensions (intel® sgx)," https://software.intel.com/en-us/sgx, Online; accessed 29-March-2019.
[10] "Introducing arm trustzone," https://developer.arm.com/technologies/trustzone, Online; accessed 29-March-2019.
[11] "Jetstream," https://browserbench.org/JetStream, Online; accessed 29-March-2019.
[12] "Kraken javascript benchmark," https://krakenbenchmark.mozilla.org, Online; accessed 29-March-2019.
[13] "Libfuzzer," https://llvm.org/docs/LibFuzzer.html, Online; accessed 29-March-2019.
[14] "Library load-order randomization," https://source.android.com/security/enhancements/enhancements70, Online; accessed 29-March-2019.
[15] "Octane," https://chromium.github.io/octane, Online; accessed 29-March-2019.
[16] "Randomizing structure layout," https://lwn.net/Articles/722293/, Online; accessed 29-March-2019.
[17] "Simulation of quantum mechanics," http://www.libquantum.de, Online; accessed 29-March-2019.
[18] "Sjeng : a chess-and-variants playing program," https://www.sjeng.org, Online; accessed 29-March-2019.
[19] "Sunspider 1.0.2 javascript benchmark," https://webkit.org/perf/sunspider/sunspider.html, Online; accessed 29-March-2019.

[20] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.

[21] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for c++," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[22] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 133–143.

[23] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *NDSS*. Citeseer, 2015.

[24] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[25] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in *Proceedings of the 2013 New Security Paradigms Workshop*. ACM, 2013, pp. 95–106.

[26] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a trap: Table randomization and protection against function-reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 243–255.

[27] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere, "Instruction set limitation in support of software diversity," in *International Conference on Information Security and Cryptology*. Springer, 2008, pp. 152–165.

[28] G. J. Duck and R. H. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 132–142.

[29] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *Symposium on Network and Distributed System Security*, 2017.

[30] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE, 1997, pp. 67–72.

[31] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization." in *USENIX Security Symposium*, 2012, pp. 475–490.

[32] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," in *Symposium on Network and Distributed Systems Security (NDSS)*, 2018, p. 148.

[33] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 571–585.

[34] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 272–280.

[35] A. Keromytis, M. Polychronakis, and V. Pappas, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 601–615.

[36] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 339–348.

[37] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "Sgxbounds: Memory safety for shielded execution," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 205–221.

[38] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*, 2015.

[39] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2009, pp. 107–126.

[40] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[41] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[42] S. Sinnadurai, Q. Zhao, and W. fai Wong, "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

[43] D. M. Stanley, D. Xu, and E. H. Spafford, "Improved kernel security through memory layout randomization," in *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*. IEEE, 2013, pp. 1–10.

[44] P. Team, "Pax address space layout randomization (aslr)," 2003.

[45] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 405–419.

[46] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers." in *NDSS*, 2015.

[47] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "Paricheck: an efficient pointer arithmetic checker for c programs," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 145–156.