Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

TC 11 Briefing Papers

Badaslr: Exceptional cases of ASLR aiding exploitation

Daehee Jang[a,*]

Sungshin W. University, Seongbuk-gu Seoul, Republic of Korea

ARTICLE INFO

Article history:
Received 30 August 2021
Revised 6 October 2021
Accepted 14 October 2021
Available online 22 October 2021

Keywords:
Address space layout randomization
memory exploit
heap randomization
low fragmentation heap
return oriented programming

ABSTRACT

Address Space Layout Randomization (ASLR) is de-facto standard exploit mitigation in our daily life software. The simplest idea of unpredictably randomizing memory layout significantly raises the bar for memory exploitation due to the additionally required attack primitives such as information leakage. Ironically, although exceptional, there are rare edge cases where ASLR becomes handy for memory exploitation. In this paper, we dig into such theoretical set of cases and name it as BadASLR. Based on our study, we introduce four categories of BadASLR: (i) aiding free chunk reclamation in heap spraying attack, (ii) aiding stack pivoting in frame-pointer null poisoning attack, (iii) reviving the exploitability of invalid pointer referencing bug, and (iv) introducing wild-card ROP gadgets in x86/x64 position independent code environment. To evaluate if BadASLR can be an actual plausible scenario, we look into real-world bug bounty cases, CTF/wargame challenges. Surprisingly, we found multiple vulnerabilities in commercial software where ASLR becomes handy for attacker. With BadASLR cases, we succeeded in exploiting peculiar vulnerabilities, and received total 10,000 USD as bug bounty reward including one CVE assignment.

1. Introduction

Modern software essentially adopts Address Space Layout Randomization (ASLR) to harden the memory from various exploitation attempts. The efficacy of ASLR is well-proven and its practicality is arguably one of the best among various software exploit mitigation techniques. Popular operating systems such as Linux, OSX and Windows by default enables/supports ASLR for their application, and even the majority of embedded software takes ASLR feature for granted.

Especially when applied to 64-bit system, ASLR makes infeasible to predict any virtual memory address from attacker's exploitation code. This forces attackers to additionally equip themselves with a stronger exploitation capability – information leakage. Thanks to ASLR, the difficulty of modern exploitation in large-scale software such as browsers and kernel has substantially increased. To exploit memory corruption bugs in such software, abusing strong information leakage bug is a must nowadays.

We emphasize that ASLR is a standard exploit mitigation technique protecting us for decades and this paper do not intend to accuse its general efficacy in any manner. However, we summarize and analyze the rare and bizarre cases which, ironically, ASLR acting as a useful tool for successful exploitation; and refer such edge cases as BadASLR[1] Under-

[1] In this paper, for simplicity, we mention heap memory layout randomization techniques as part of ASLR in general. To be more

* Corresponding author.
E-mail address: djang@sungshin.ac.kr
https://doi.org/10.1016/j.cose.2021.102510
0167-4048/

standing such BadASLR cases will advance the completeness of knowledge and provide thought provoking insights in future research.

There are four types of BadASLR categorized in this paper:

- Type-I: Supporting heap layout manipulation for use-after-free and heap overflow.
- Type-II: Stack-pivoting support in frame pointer null-poisoning.
- Type-III: Reviving exploitability of invalid pointer.
- Type-IV: Introducing wild-card ROP gadgets with diversified branch offset encoding.

BadASLR Type-I is cases related to randomization in heap chunk allocation timing and their layout adjustment in attacker's advantage. Technically, the term ASLR indicates randomizing the address/order of memory segments at page granularity, but in this paper we include ASLR for all types of memory layout randomization including heap chunk positioning (we clarify it in Section 2). BadASLR Type-II is related to special type of stack-based buffer overflow that allows null-poisoning against saved stack frame pointers (e.g., RBP register value in x64). This type of buffer overflow can also occur in heap. Exploitation of such partial overwrite involves pivoting/lifting an existing pointer. BadASLR Type-III is related to randomization in `mmap` (page allocation) or dynamic library loading. The abusing scenario for Type-III is extremely unlikely in 64-bit address space environment but quite plausible in 32-bit virtual address space application. Finally, BadASLR Type-IV is conceptually far from other BadASLR cases. The scenario suggests that an ironic case which ASLR introducing more diverse ROP gadgets is possible due to the randomized inter-segment distance. Depending on compiler/linker options, branch target addressing in position independent code might change its instruction encoding due to ASLR; which gives diversity in encoded branch target offsets. This issue only affects Intel CISC instruction set architecture where the instructions can split with byte granularity; thus any byte sequence inside instruction can be an ROP gadget.

Each BadASLR cases are first theoretically discussed based on assumptions and demonstrated with Proof-of-Concept codes/examples. Afterward, based on our theoretical analysis of BadASLR, we study real-world memory corruption exploits and CTF/wargame challenges to see if such ironical cases can actually happen in practice. Surprisingly, we found multiple real-world cases for BadASLR Type-I and Type-III. Four bugs we discovered was only possible to exploit it with the help of ASLR. In particular, we found a peculiar heap overflow vulnerability in KMPlayer video parser and successfully exploited the bug with CVE assignment (CVE-2018-5200). We exploited multiple BadASLR cases and got approximately 10,000 USD bug bounty rewards in total.

The rest of this paper continues as follow: we provide background knowledge regarding basic memory corruption exploits and clarify our premise and assumptions in Section 2. Detailed description and theoratical explanation of each BadASLR cases are discussed in Section 3 with proof-of-concept implementation code. As evaluation, we search various bug-bounty cases and CTF/wargame challenges to check if our BadASLR cases are plausible and could be a real-world scenario in Section 4. In Section 5, we discuss issues related to BadASLR. In Section 6 we discuss related works, and conclude in Section 7.

precise, ASLR is more specific term for randomizing the location of memory segments such as.text or library mapping.

2. Background and assumptions

2.1. Memory exploitation related terms

Free Chunk Reclamation. Data objects are dynamically allocated to the heap as needed, and should be freed when they are no longer required. However, a programmer could create some logic that accidentally references the pointer of a freed object and uses the object as if it were still allocated. This is famously known as a *use-after-free* bug, and the pointer pointing to the freed object is called Dangling Pointer. *Free chunk reclamation* is an exploitation attempt for re-allocating the memory space pointed by dangling pointer. Upon successful free chunk reclamation with attacker-controlled data, application uses the attackers controlled data as if it is previously allocated object, which is totally different from attacker's data. Therefore, if the original object contains function pointer which decides the program execution flow, attacker could hijack such pointer value to manipulate the program execution.

Heap Overflow. Heap overflow refers to a buffer overflow vulnerability that occurs inside the heap area. There could be many reasons that cause this vulnerability. One of the common case [12] is: (i). programmer accidentally declares the length variable of buffer as signed integer type, (ii). attacker uses a negative number in order to mislead the buffer length calculation, (iii). attacker fills up the buffer with an unlimited amount of data. Since the heap layout is dependent on the memory allocator implementation and application semantics, any heap overflow exploitation strategy would be application-specific. However, heap overflow bug can be commonly exploited in two ways: (i). corrupting the metadata of the memory allocator or (ii). corrupting the applications heap data adjacent to the buffer. The heap-overflow exploit discussed in this paper mainly refers to the latter case, whereby the application heap data is corrupted, not the metadata of memory allocator.

NULL Poisoning. NULL poisoning is a specific type of buffer overflow where the overwritten value can only be NULL. Typically, this is also refered as *off-by-one* error where the maximum length of accessible array is one byte greater than the buffer size. Because of the confusing array indexing (e.g., array index starts with zero, but human starts counting number with one), null poisoning caused by off-by-one error is quite common in practice.

2.2. ASLR And heap randomization

Technically, ASLR is not a precise term for heap randomization as there are multiple aspects in randomizing general heap memory layout. In general, ASLR for heap signifies randomizing the base address of heap segment. However, for sim-

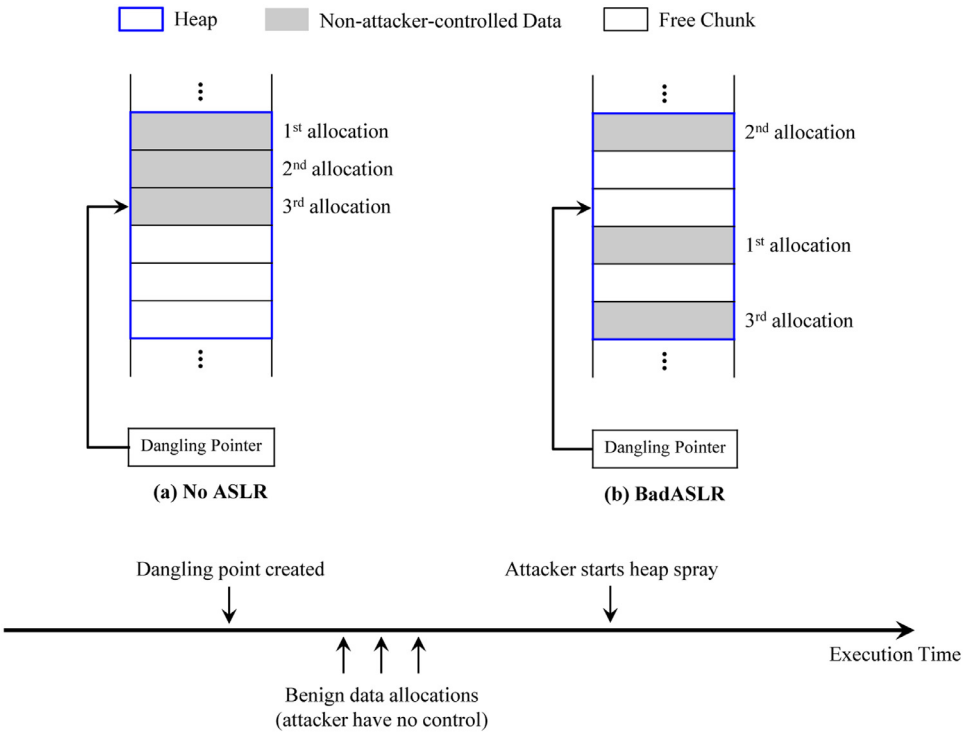

Fig. 1 – BadASLR-(i) turning unexploitable use-after-free situation into highly exploitable situation. The figure is illustrating the Fig. 3 proof-of-concept code.

plicity, we use the term ASLR for all types of heap randomization including the non-deterministic free chunk allocation Novark and Berger (2010). Any endeavor for randomizing the memory address layout (including relative positioning) is referred as ASLR in our paper.

2.3. *Low fragmentation heap*

Because the case studies in this paper often involves windows Low Fragmentation Heap (LFH) allocator usage, we describe its basic background. The LFH memory allocator was introduced in Windows 2000 and XP. As the name suggests, LFH reduces external memory fragmentation by gathering similar-sized chunks adjacent to one another. In LFH, each heap bucket (group of similar-sized chunks) contains hundreds or thousands of chunks. All chunks inside the same bucket are of the same size. When the application requests memory allocation of size N, similar to most memory allocators, LFH rounds up the N to a multiple of eight, and searches for an available slot in the existing free chunks for the calculated size. If there is no available free chunk for the expected allocation size, the allocator creates a new heap bucket. Although LFH reduces external memory fragmentation, it also induces the internal fragmentation of the heap bucket. However, if the application uses the heap for a large amount of data, the proportion of internal fragmentation will be quite small. Considering that most applications use an enormous amount of heap, LFH represents an effective choice for reducing the overall memory fragmentation. The use of LFH can be explicitly configured by both users and developers. However, in general, using the LFH feature is the default configuration for most Windows-based software such as widely-used web browsers and document processing applications.

2.4. *Position independent code*

Position Independent Code (PIC) is a code fragment that can be loaded to any memory address. To fully apply ASLR without any predictable memory segment, it is essential to compile codes in a form of PIC because their loaded address is unpredictable due to ASLR. The main difference of PIC code and non-PIC code is the offset encoding of branch target address. Without ASLR, non-PIC code could use an instruction such as `jmp 0x8048123` using the branch target as 32-bit absolute address encoded inside the instruction. However, in PIC code, using such absolute memory address is infeasible because no memory address is decided before program run-time. For absolute address based branch instructions of PIC code, dynamic linker is responsible to resolve/update such address at run-time. This behavior is typically observed in Linux kernel modules. We refer PIC code while discussing BadASLR Type-IV.

2.5. *Return oriented programming*

Return Oriented Programming (ROP) is a practical exploitation technique often used in various memory attacks which allows the adversary to circumvent the Data Execution Prevention (DEP). The basic idea is to re-use existing codes. The essence is utilizing the `ret` instruction which takes next instruction pointer from stack memory. If attacker can put malicious/controlled data into stack or hijack stack pointer to their

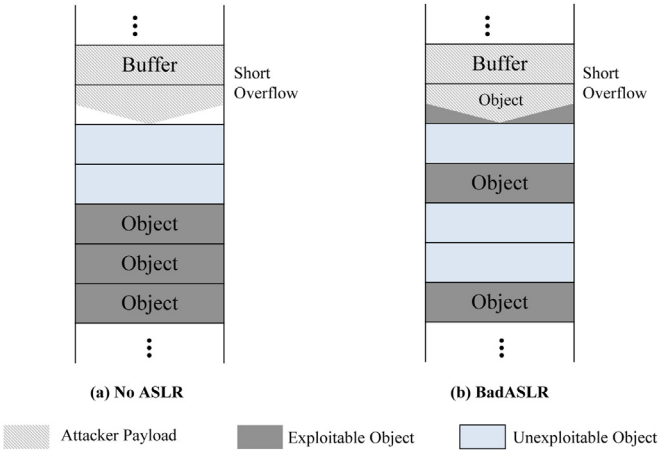

Fig. 2 – BadASLR-(i) turning unexploitable heap overflow vulnerability into exploitable one. The scenario is a conceptually same to use-after-free case with only minor change in exploitation scenario setup.

fake stack payload, an attacker can chain the return instructions followed by piece of code from existing code segment. Gathering such small piece of codes and stitching them can ultimately allow an attacker to execute an arbitrary code. This technique is also referred as code-reuse-attack.

3. Design

In this section, we categorize BadASLR into four types and explain their details in theory.

3.1. *Badaslr type-I: Aiding free chunk reclamation*

Heap spray is frequently utilized to enhance the reliability of memory corruption based exploit. There are multiple data types to spray inside heap depending on the exploitation environment. Although its outdated, the basic data type for heap spray is the *NOP-sled* (and shellcode). Spraying the NOP-sled is only meaningful when the target application lacks Data Execution Prevention (DEP) (Shacham, Hovav, 2007). Another type of data attackers typically spray inside heap is *objects* embedding pointers. These objects are sprayed in order to place at least one of them at proper memory position (e.g., free chunk reclamation). In theory, it is possible that ASLR (technically, it is called heap randomization but we refer it as ASLR to simplify terms) helping attacker to this end. We refer such counter-intuitive situation as BadASLR Type-I.

For example, in use-after-free, consider a hypothetical scenario where an application happens to allocate a chunk of pure data (e.g., image) immediately after a dangling pointer is created as illustrated in Fig. 1. In such case, use-after-free vulnerability becomes unexploitable as the inevitable execution flow immediately re-reclaims the inadvertently freed (dangling-pointed) chunk.

Same principle can be also applied to heap overflow vulnerability case. In the figure Fig. 2, we can consider the adjacent heap chunk within a overwrite-able range as inadvertently freed chunk of the use-after-free case. If a program subsequently makes allocation which consumes such memory region with uncontrolled junk data (e.g., version string, constant numbers) before attacker takes over heap control, exploitation will become infeasible. Fig. 3 is a proof-of-concept example C code for this scenario. In the example code, dangling pointer is created at line 15. To exploit this as use-after-free, attacker must reclaim this dangling-pointed free chunk with his/her controlled data (e.g., byte stream which attacker injected as part of input). However, in line 18 - 23, example program allocates three heap chunks that has equal size to the dangling-pointed free chunk. Without ASLR, the use-after-free becomes impossible to exploit because attacker cannot reclaim the target free chunk. But with ASLR, free chunk allocation ordering

```
1  // Object embedding pointer to read/write memory
2  typedef struct _tagOBJ{
3      int id;
4      char* read_write_ptr;
5  }OBJ, *POBJ;
6  // Pure constant-based data object
7  typedef struct _tagOBJ2{
8      int version1;
9      int version2;
10 }OBJ2, *POBJ2;
11 void BadASLR_Example(){
12     // program allocates pointer-embedding object
13     POBJ p = (POBJ)malloc(sizeof(OBJ));
14     // ...
15     free(p);    // inadvertent free!
16
17     // program allocates three objects (attacker have no
        control)
18     POBJ2 p1 = (POBJ2)malloc(sizeof(OBJ2));
19     POBJ2 p2 = (POBJ2)malloc(sizeof(OBJ2));
20     POBJ2 p3 = (POBJ2)malloc(sizeof(OBJ2));
21     memset(p1, 0, sizeof(OBJ2));
22     memset(p2, 0, sizeof(OBJ2));
23     memset(p3, 0, sizeof(OBJ2));
24
25     // attacker can spray heap at this point
26     parse_input( ... );
27
28     // use-after-free!
29     *(p->read_write_ptr) = data;
30 }
```

Fig. 3 – Proof-of-concept code for BadASLR Type1.

becomes random, thus it is unlikely that dangling-pointed target chunk will be consumed by three subsequent allocations. As a result, attacker gets a chance to reclaim the chunk at line 26 and exploit this use-after-free bug.

We note that this scenario is more plausible for relatively simple application such as file parser (video, image, etc) that has limited user-application interface. We also found a partial exploitation step in 64bit Edge browser exploitation (CVE-2016-0191 cve (2016)) falls into BadASLR Type-I. However, in general, when it becomes to 64bit browser/kernel exploitation, BadASLR Type-I scenario is very unlikely to be a real thing because the complexity of heap data management and the interaction channel between application and user becomes gigantic.

3.2. *Badaslr type-II: Aiding stack pivot in frame pointer NULL-Poisoning*

As a specific case of off-by-one bug, null-poisoning is common in heap exploitation. For example, a single byte of metadata (e.g., size) corruption can lead things into chaos how (0000). This technique can be equally applied to stack data structures. In fact, if stack canary is bypass-able or absent, off-by-one null-poisoning is a promising/reliable exploitation primitive to pivot the stack frame to initiate ROP attack.

One might ask why we need to pivot the stack before ROP. Consider if the stack buffer overflow is based on null-terminated ascii string. Building ROP chain with such restriction is often impossible because pointers usually require non-ascii bytes. In this situation, an alternative exploitation strategy is pivoting the stack where attacker already prepared the ROP chain.

Because most of the compilers insert stack frame's base address (e.g., the value of RBP register in x64) on the stack and chain them up for each function calls, even partially corrupting such saved frame pointer can pivot the stack and overlap the return address with attacker's local variable in other function frames (let us assume these local variables can be a working ROP chain). By null-poisoning the stack frame pointer, the parent function stack frame will pivot towards lower memory address up to 255 (or 65,535 for lower two bytes poisoning) bytes thus can overlap with attacker's local variable in other stack frame. Fig. 4 illustrates this scenario in detail.

The essence of BadASLR Type-II is that because modern ASLR changes stack base address and offset every time, the offset between pivoted stack frame and attacker's local variable (which aims to overwrite parent function's return address) can randomly change across executions within a predictable range of 255 or 65,535 bytes. Therefore, with average 255 (or 65535) trials of exploitation attempt, attacker can eventually overlap his/her local variable at a proper ROP chain position regardless of the local variable's relative positioning inside the stack frame. Basically, in this hypothetical scenario, ASLR is converting the *NULL byte poisoning* primitive into more useful *random byte poisoning*. Without ASLR, NULL byte poisoning in this theoretical setup would be always exploitable with single exploitation attempt, or never exploitable if stack layout turns out unlucky. With ASLR, for every attack trial, attacker has small chance to succeed exploitation regardless of how the stack layout is positioned (as ASLR provide diversity in stack frame layout).

3.3. *BadASLR-(iii): Reviving invalid pointer reference*

Invalid pointer refers to a virtual memory address with no accessible corresponding memory segment. Virtual memory address is typically composed with stack, heap, code, and data segment with different memory access permissions. Additionally, for dynamically linked binaries, shared library images (or other files) are loaded into memory via program interpreter (e.g., ld-linux.so) dynamically.

Now let us consider a hypothetical bug which allows an attacker to reference a fixed constant (say, `0x12345678`) as a pointer of an object (confusing constant and pointer). If the virtual address `0x12345678` has no valid segment mapping, without ASLR, this bug is never exploitable. However, ASLR opens a possibility for turning this bug into something exploitable with small chance. Because of ASLR, the address `0x12345678` is no longer guaranteed to be inaccessible considering ASLR moving memory segments including library mapping and other dynamically mapped segments (e.g., `MapViewOfFile` in Windows). This is unrealistic in 64bit address space, however, in 32bit address space the entropy of segment base address is quite small. Surprisingly, we found an actual real-world case of this unlikely exploitation scenario. We visualize this exceptional hypothetical scenario in Fig. 5 and discuss real-world case in Section 4.

3.4. *BadASLR-(iv): Introducing wild card ROP gadget*

In theory, ASLR might contribute to increase the diversity of ROP gadgets in x86/64 position independent binaries such as shared objects. Position independent codes must handle out-of-segment branches such as imported library calls. Because the relative distance between position independent code and library call target is unknown at compile time and decided at runtime, branch offset for such call instructions must be updated at runtime as well. Usually, such calls utilize additional data structures known as procedure linkage table (PLT) and global offset table (GOT) to calculate the target address using a dynamically updated function pointer. However, depending on the compiler options, dynamic linker can also resolve such offset by updating the instruction code at runtime. In the latter case, ASLR introduces wild card ROP gadgets as a part of branch target offset encoding.

In Fig. 6, the branch target offset in line 7 and 18 is changed by ASLR at runtime. Because Intel is CISC machine, the changing bytes can act as a *wild card ROP gadgets* which can become any instruction attacker expects with some attack iteration. In fact, there are some crucial ROP gadgets composed only with 2 bytes such as stack pivot (e.g., xchg eax, esp;ret as `94 c3`). Admittedly, exploitation only viable with BadASLR-IV is unrealistic for some reasons: (i) it is likely that such small gadgets would be already available somewhere else in the code base (e.g., encoded constants or fixed relative offset) even without BadASLR primitive, (ii) in reality, there are multiple ways to construct ROP chains; thus an exploitation scenario only possible via BadASLR-IV is unrealistic. However, we find it is educative and interesting to explore such theoretical scenarios.

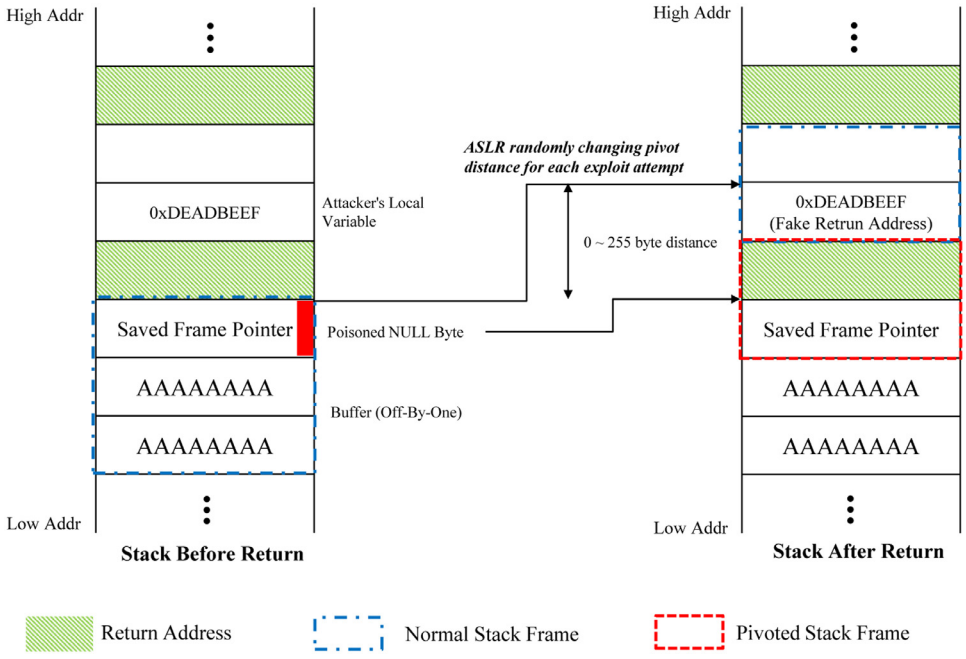

Fig. 4 – BadASLR-(ii) turning off-by-one NULL byte poisoning against stack frame pointer into random byte (from 0 to 255 range) poisoning.

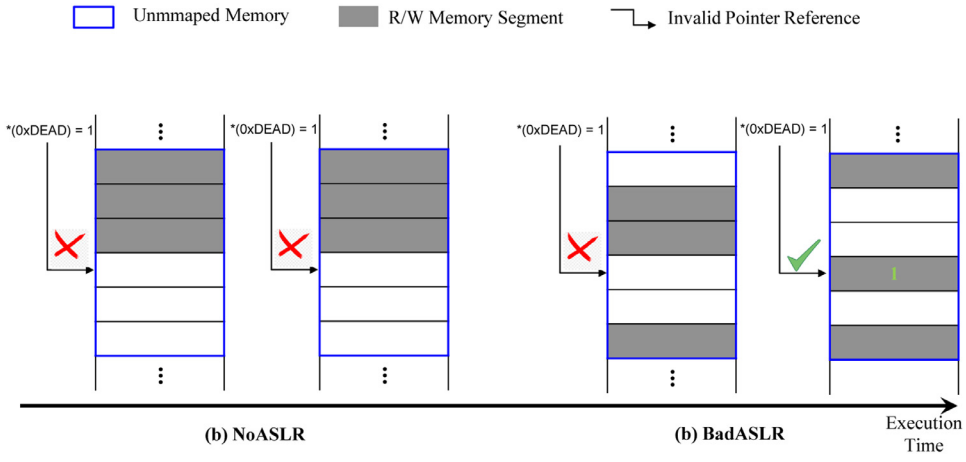

Fig. 5 – BadASLR-(iii) giving a survival chance to invalid pointer reference. Totally unexploitable invalid pointer reference might become useful (with small chance) under ASLR.

4. Evaluation

To find out the prevalence of BadASLR, we analyze various exploitation cases. Based on our study, we report that BadASLR is extremely unlikely to be an exploitation primitive in large-scale, user interactive application such as browsers and kernel. However, we found that it can be a working exploitation primitive in small/non-user-interactive application such as multimedia/document parser.

4.1. *Case study: Heap overflow in WPS writer*

WPS is an office suite software developed by Kingsoft kin (0000). We have discovered a heap overflow bug from WPS Writer exploitable with BadASLR Type-I. While initializing data for a special object (let us denote as SOBJ) in specific condition, the program accidentally calculates the size of the object twice bigger than it should be. As a result, the program initializes memory contents of SOBJ with a miscalculated size and corrupts adjacent heap region. As a result, attacker can overwrite adjacent heap region with limited (fixed) length with data such as color code, font size such that attacker can specify in the input file. Fig. 7 is the example C code for explaining this bug.

In file parser exploitation, there is no leeway over the heap allocation timing and overall heap memory management (de-allocation, re-allocation, etc) because there is no continuous interaction between attacker's data and program. Unlike JavaScript exploitation, attacker cannot freely decide *when* to

```
// 1st execution with ASLR
Disassembly of foo
0000000000000000 <foo>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 8d 3d 30 04 00 00  lea    0x430(%rip),%rdi
   b: e8 c0 /*63 52*/ 7f    callq  <puts>
  10: 90                    nop
  11: 5d                    pop    %rbp
  12: c3                    retq

// 2nd execution with ASLR
Disassembly of foo:
0000000000000000 <foo>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 8d 3d 30 04 00 00  lea    0x430(%rip),%rdi
   b: e8 c0 /*43 69*/ 7f    callq  <puts>
  10: 90                    nop
  11: 5d                    pop    %rbp
  12: c3                    retq
```

Fig. 6 – Proof-of-concept example for BadASLR Type4.

```
typedef struct _tagSOBJ{
    unsigned int color;
    unsigned int font;
}SOBJ, *PSOBJ;
void Parse_Document(){
  ...
  PSOBJ p = (PSOBJ)malloc(sizeof(SOBJ));
  p->color = get_color();
  p->font = get_font();
  ...
  // blocks exploitation under no-ASLR
  alloc_useless_data();
  ...
  if( special_condition() ){
      // bug (heap overflow)
      p++;
  }
  ...
  p->color = get_color();
  p->font = get_font();
  }
  ...
```

Fig. 7 – Example C code to explain WPS bug.

allocate/de-allocate object. Therefore, if the adjacent object (that attacker can overwrite via bug) happens to be something useless, exploitation becomes infeasible.

In fact, in Windows 7 environment where LFH heap lacked ASLR primitive (e.g., allocation sequence is deterministic), the bug we discovered was never exploitable because the parser immediately allocates a useless data chunk at the precise position where attacker can overwrite (adjacent to SOBJ). Therefore, even attacker triggers buffer overflow, there is no meaningful target to overwrite. However, ironically, this bug became exploitable in Windows 8 environment where LFH adopted ASLR primitive in their chunk allocation policy. As free chunk for allocation is randomly chosen, it is unlikely that the useless data (immediately allocated after SOBJ allocation) will consume the *overwrite-able memory* before the execution flow reaches attacker's heap control.

Table 1 – Average heap spray amount to reclaim a specific free chunk in randomized LFH heap bucket. N is the number of initial allocation, K is the percentage of randomly de-allocated chunks, SD stands for standard deviation. The average value is calculated based on 10 iteration.

Chunk Size	N	K	Average Spray Amount	SD
50	1,000	20	216.8	61.3
		50	251.1	131.5
		80	270.0	231.9
	10,000	20	1559.8	705.0
		50	2611.0	1206.1
		80	3967.0	1879.1
	100,000	20	11423.0	3464.0
		50	14831.1	6021.3
		80	13533.4	9790.0
500	1,000	20	201.8	26.8
		50	158.6	104.0
		80	219.6	127.8
	10,000	20	1196.3	617.2
		50	3037.6	1044.4
		80	3360.8	1896.5
	100,000	20	8350.2	4125.8
		50	12695.1	5971.3
		80	15300.9	6719.8

Due to the BadASLR Type-I, it is highly likely that the overwrite-able heap region remains as a free space until attacker takes over control for heap allocation. Although we cannot control *when* to allocate our data, we had control over *how many*; which allow as to spray the heap to some extent. By adjusting proper heap-spray amount, with high probability, we were able to overwrite useful (in terms of exploitation) objects embedding function pointers. Table 1 is an experiment result for finding minimal heap spray amount in randomization-enabled Windows 8.1 LFH heap. We allocate N chunks continuously, then randomly de-allocate K percent of them. Afterward, we randomly select a target free chunk (assuming as if dangling pointed, or overwrite-able) and spray the heap until our target gets reclaimed. The result suggests that we can practically reclaim the target free chunk despite of the randomized allocation sequence as heap defragments. We chose chunk size 50 and 500 based on heap memory analysis that such sizes are most prevalent as objects. From the experiment, we can see that object size do not mainly affect the free chunk reclamation process. The result also suggests that required spray amount is usually proportionate to N. However, when N is 100,000 and chunk size is 50, average spray amount is higher when K is 50 than 80; which seems counter intuitive but it is plausible if we consider multiple factors such as heap bucket size.

4.2. *Case study: Heap overflow in KMPlayer*

Similarly to WPS Write exploitation, we discovered BadASLR Type-I case in KMPlayer as well. We reported this bug to vendor and got 4500 USD as reward and also a CVE assignment (CVE-2018-5200 cve (2018)). In this vulnerability, KMPlayer do not

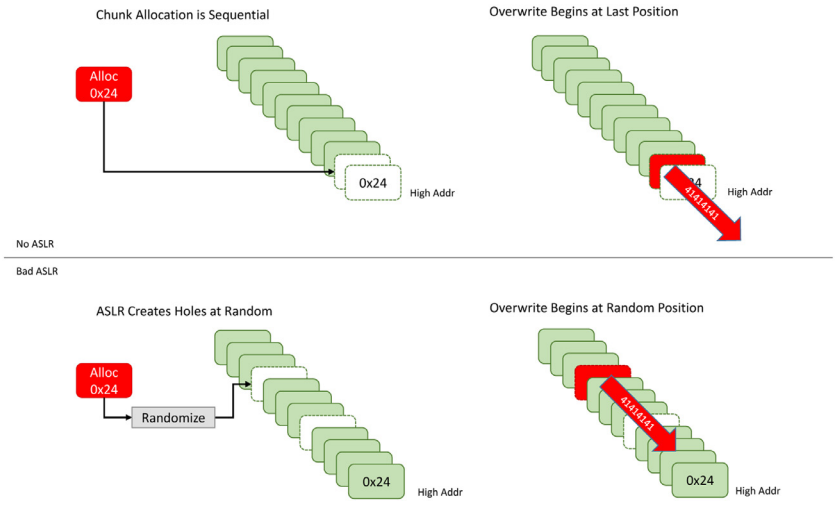

Fig. 8 – ASLR helping heap buffer overflow to overwrite a target object with high probability in CVE-2018-5200 KMPlayer exploitation.

consider the edge case of H.263 video packet decoding (H.263 Sorenson Type). As a result, attacker can trigger heap overflow with crafted H.263 Sorenson encoded video. Attacker had full control over the length of attack payload, however, because the overflow-ing data is decompressed video stream, we had to overwrite the memory only with decompressed pixel values which gave us significant restriction in memory corruption. To survive the lossy decompression, exploit payload is mostly composed with low-entropy byte patterns. Ultimately, we were able to trigger heap buffer overflow inside heap buck of size 0x24 chunks. To exploit this, we spray an object of size 0x24 with C++ virtual table pointer. After spraying the heap bucket with such an object, we also allocate the decompression buffer of exact same size (0x24); then trigger the buffer overflow.

Unfortunately, because the decompression (which triggers buffer overflow) can only be triggered *after* we spray the size-0x24 heap bucket, it was not possible to overwrite the target object (that has virtual table pointer) if the heap chunks are sequentially allocated towards higher address in deterministic order. However, when ASLR is applied to heap chunk allocation algorithm, we could reliably overwrite our target object by precisely controlling the amount of heap spray and create an hole with couple de-allocation. Fig. 8 illustrates this effect. Because of such exceptional restriction in memory allocation order, ASLR effectively supports the exploitation by allowing attacker to create a hole in the middle of the sprayed objects.

4.3. *Case study: Use-After-Free in HWP parser*

One of a use-after-free bug we have discovered from a document parser had BadASLR Type-I case. Fig. 9 is a simplified pseudo code of the erroneous parsing logic resulting use-after-free. From the pseudo code, we can observe that when the parser encounters an invalid object, it frees the object and escapes the parsing loop. However, afterward, the program overlooks the prior exceptional de-allocation and reference VPTR

```
1  // pseudo-code for parser logic
2  void Parse_Document(){
3    ...
4    while( read_file() ){
5      ...
6      object = allocate();  // attacker controls allocation
7      if( !isvalid(object) ){
8        free( object )
9        break
10     }
11     parse( object )
12     ...
13   }
14   process_GUI();    // no control for attacker
15   ...
16   process_data();   // attacker re-gains heap control
17   ...
18   object.func1();   // use after free
19   ...
```

Fig. 9 – Example C code to explain HWP bug.

(virtual pointer for C++ virtual table) pointer as if the object was never freed.

We can also see that immediately after exiting the main parsing loop, the program starts processing GUI elements. After the GUI processing, the logic continues and finally references the prior dangling pointer. We note that the execution flow of this logic is deterministic thus attacker has no control over it. Unfortunately for the attacker, image raster data allocation from ”process_GUI()” step involves allocating heap chunk that has exactly the same size as the dangling pointed object. Therefore, without the randomization effect in heap chunk allocation, the use-after-free in this example becomes impossible to exploit. However with BadASLR Type-I, with high chance, memory allocation in ”process_GUI()” will not immediately reuse the dangling pointed free chunk. Ironically, we were able to reliably exploit this bug and hijack the control flow in Windows 8 LFH environment (randomized allocation), but could not exploit the bug under Windows 7 or XP which

```
1  // required -fno-stack-protector and -m32
2  // BASE = ((void*)0x80000000)
3  int is_ascii(int c){
4    if(c>=0x20 && c<=0x7f) return 1;
5    return 0;
6  }
7  void vuln(){
8    int a[40];
9    strcpy((char*)a, (char*)BASE);
10 }
11 void main(int argc, char** arg, char** env){
12     if(argc!=1) return;
13     int i, j;
14     for(i=0; env[i]; i++) for(j=0; env[i][j]; j++) env[i][
        j]=0;
15   char* res = mmap(BASE, 4096, 7, MAP_ANONYMOUS |
        MAP_FIXED | MAP_PRIVATE, -1, 0);
16   if(res != BASE){
17     printf("mmap failed. tell admin\n");
18     _exit(1);
19   }
20   printf("Input text : ");
21   unsigned int n=0;
22   while( n<400 && is_ascii(res[n++]=getchar()) );
23   printf("triggering bug...\n");
24   vuln();
25 }
```

Fig. 10 – Example Wargame Challenge for BadASLR Type-II.

do not support heap chunk randomization in their LFH allocation. We reported this bug to vendor to fix it and got compensated with 2500 USD.

4.4. *Case study: Synthesized examples*

We failed to discover BadASLR Type-II case in real-world exploitation examples, however, we were able to synthesize a simple program which falls into this category. The vulnerability in this program is based on strcpy with printable ascii-only payload. Fig. 10 is the source code of this toy program. In `vuln` function, there is a 40 byte stack-based buffer. In line 9, `strcpy` copies printable ascii-only string from memory segment located at 0x80000000 hence far from stack and consisted with non-ascii bytes only (e.g., 0x80 and NULL). Ultimately, the only way to exploit this bug is NULL poisoning (partial overwrite) the stack frame pointer using the NULL termination byte of the printable-ascii string[2]. Without ASLR, this challenge is infeasible to solve because the pivoted stack will never properly overlap to attacker-intended relative address. Admittedly, this is an artificial vulnerability and made up exploitation environment. However, the example proves that BadASLR Type-II is a theoretically working scenario.

4.5. *Case study: Invalid pointer reference in HWP parser*

We found a use-after-free in HWP document parser while parsing V3 file format. Because V3 file format do not support various document components, allocating/controlling heap data is very limited. Therefore, although we found use-after-free, it was impossible to reclaim the dangling pointer with malicious data that we can control. As a result, the only choice for abusing this use-after-free bug is making the heap allocator to overwrite the dangling pointed region and corrupt data.

Fortunately, the dangling pointed object had virtual table pointer (VPTR) as first member variable thus we could corrupt the lower 2 bytes of VPTR pointer with *next offset* metadata which the LFH allocator manages. In this case, the *next offset* was changed into 65535 which indicates the next chunk is out of range boundary. As a result, we could reference VPTR pointer who's lower 16bit is always corrupted with 0xFFFF. Fig. 11 is the heap memory dump of this situation. Because VPTR is included as part of *text* section, the 0xFFFF overwrite shifted the pointer to point read-only literal constant 0x52691004 included in the same segment. Therefore, the initial use-after-free turned into *invalid pointer reference* bug. Without ASLR, there was no valid segment mapping at this address. However, because of ASLR we had a small chance (1 out of hundreds) to occasionally reference this pointer to finally execute our controlled heap memory region as function; thus allowing shellcode execution in no-DEP environment. This is an example case of BadASLR Type-III.

4.6. *Case study: Wild card ROP gadgets*

Unfortunately, we could not find example case for BadASLR-IV. However, we report that branch target encoding instructions being affected by ASLR is commonly observed in production software.

There are multiple ways to resolve dynamic branch target address (e.g., combination of procedure linkage table and global offset table). Depending on the application, a system could simply re-write the branch offset at runtime which introduce an ROP gadgets do not appear via static binary analysis. For example, we can observe such behavior in Linux Kernel Modules (LKM) insertion. Because modules are dynamically inserted to kernel memory and there insertion sequence can change, the branch offset of kernel imported functions can also change across rebooting. The unpredictability of such offset becomes higher if we consider KASLR kas (0000). Fig. 12 illustrates the branch offset relocation in LKM loading. In the figure, the branch offset of `mcount` and `strstr` function (boxed with red line) is changed after the module inserts into kernel memory.

However, most of the recent compilers for building majority of user application uses PLT/GOT as default approach to handle dynamic linking. Therefore, although it is theoretically possible that ASLR can introduce more ROP gadgets due to diversified branch target offsets, it shouldn't be very common. We have summarized our overall case study evaluation in Table 2.

5. Discussion

5.1. *ASLR And information leakage*

When an application supports interactive scripting (e.g., JavaScript), information leakage bugs could allow an attacker to dynamically calculate and figure out the memory address of important data structures on-the-fly in exploitation. To

[2] Overwriting the lower bytes of partial pointer and pivoting/lifting the address is a quite common vulnerability exploitation technique Kikuchi and Arimizu (2014)

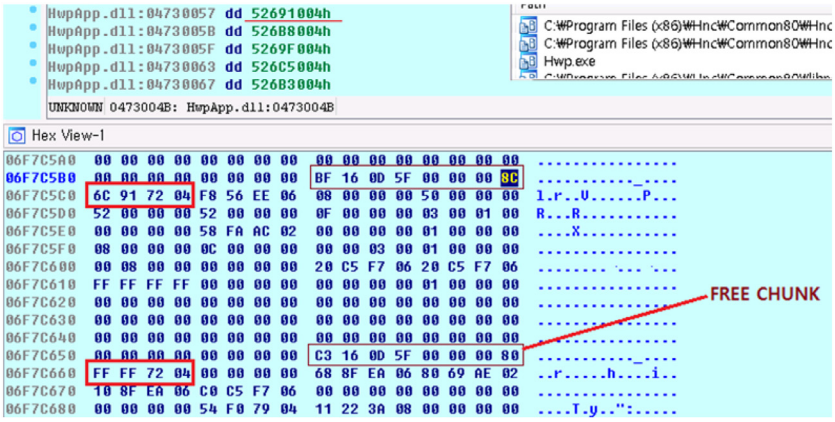

Fig. 11 – Lower 16 bit of VPTR pointer (originally 0x472916c) is overwritten by 0xffff (0x472ffff at 0x6f7c660). Corrupted VPTR pointer deterministically points an invalid pointer `0x52691004` **which is a fragment of pure data.**

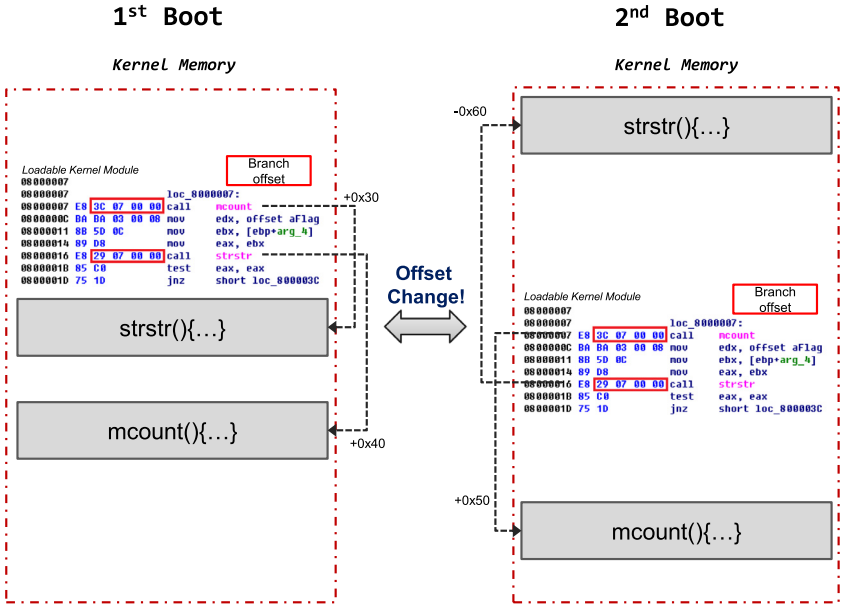

Fig. 12 – Branch offset relocation across booting in Linux Lodable Kernel Module (LKM). Because kernel modules are dynamically loaded, the branch offset between an import function's source and destination offset can be decided at loading time. This also can change due to KASLR.

Table 2 – Summary of Case Study Result. BadASLR-(i) is quite common in 32-bit file parser applications.

BadASLR Type	Prevalence Research Result
Type-I	Found three cases in real-world application
Type-II	Not Found
Type-III	Found one case in real-world application
Type-IV	Not Found

fully break ASLR, the best information leakage vulnerability is the arbitrary memory read capability. For example, when reading an array element via indexing, if the index is not checked for its boundary, it is possible to read the entire memory content with arbitrary index value. However, ASLR bypassing is also feasible by revealing only a single pointer value. Upon the leakage of a single pointer, all memory content bounded to the same segment is leaked. This is because attacker can obtain any address in the same segment by adding/subtracting pre-calculable offsets between the leaked pointer address and target address (as relative distance is constant). However, leaking a pointer bounded to a specific memory segment do not also reveal other segments affected by ASLR.

5.2. *Exploitation in non-interactive software*

BadASLR is set of edge cases thus uncommon in practice. However, according to our evaluation, the prevalence largely

depends on interactive-ness of the target application. If the application is non-interactive – exploitation cannot change its logic dynamically upon feedbacks – there are some real-world cases exhibiting BadASLR. We estimate BadASLR will be extremely rare in browser/kernel as such software is fully interactive; however fairly common in multimedia processor, document parser, and so forth.

5.3. *Wild card ROP gadgets*

We define wild card ROP gadget as a fragment of instruction which randomly change across execution. Previously, a number of studies explored the prevalence of ROP gadgets, automation for finding them, and ideas to reduce them Coffman et al. (2016); Davi et al. (2014); Follner et al. (2016); Mortimer (2019); Stancill et al. (2013). To that end, in theory, we demonstrated that ASLR introduces wild card ROP gadgets (BadASLR Type-IV) in Intel ISA because branch instructions in position-independent-code can dynamically change their embedded offset encoding due to ASLR. According to our evaluation, this turns out only plausible as theory.

6. Related work

6.1. *Good system introducing new bugs*

Wressnegger et al. demonstrated in their paper (Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms) that migrating a 32-bit application into 64-bit environment could introduce new bugs Wressnegger et al. (2016). Such bugs are mainly caused by the confusing interpretation of `LONG` type variable which is considered 32bit in Windows however treated as 64bit in Linux environment. The paper found various example cases of such errors and evaluated the prevalence. We also present the paper in a similar sense but with different topic: edge cases of ASLR supporting exploitation.

6.2. *Fine-Grained ASLR and badaslr*

ASLR we mention in this paper is based on real-world deployed coarse-grained version. However, recent works are proposing fine-grained ASLR Davi et al. (2013); Hiser et al. (2012); Kil et al. (2006); Li et al. (2010); Seo et al. (2017); Snow et al. (2013); Wartell et al. (2012). Instead of applying address randomization to memory segments, fine grained ASLR pursuits randomizing location of basic blocks. Obviously, such attempt will incur additional overhead and complication in return of security efficacy. Under the fine-grained ASLR assumption, BadASLR theory remains equally effective. In fact, BadASLR Type-IV becomes even more plausible under fine-grained ASLR because the branch offset always change across the execution.

6.3. *Non-Randomization based defense*

Exploitation discussed in this paper is mainly based on heap vulnerabilities such as use-after-free dangling pointer and ASLR is effective mitigation in general. However, there are other types of vulnerabilities and heap exploit mitigation which is orthogonal to ASLR as well. For example, to detect the exploitation of dangling pointers, dynamic analysis approaches Caballero et al. (2012); Lee et al. (2015); Nagarakatte et al. (2010); Serebryany et al. (2012); Xu et al. (2004); Younan (2015) utilize metadata that contain the status/relation between objects and the corresponding pointers. However, compared to ASLR-like randomization, it is challenging to maintain precise metadata under low performance degradation. Tracking numerous pointers and their propagation incurs high performance overhead, which prevents such approaches from being widely adopted in complex programs.

Isolation based heap protection is also an effective heap defense orthogonal to ASLR. The isolation heap protection approach separates heap allocation area for each object type. In conventional applications, all objects are present in a single shared heap area, so that the use-after-free and heap overflow attacks are more feasible. On the other hand, the isolation heap scheme can mitigate the use-after-free and heap overflow attacks by assigning the isolated heap allocation space for each individual object, thereby removing the possibility that the attacker-targeted heap object (e.g., dangling-pointed free chunk) overlapped with the heap object under the control of the attacker. Cling Akritidis (2010) shows a nice work regarding this approach.

6.4. *Various randomization approaches in heap chunk allocation*

Heap randomization and its exploit is the main issues covered in our evaluation. We only focused on heap randomization which shuffles the allocation order in free chunk selection which was deployed in real-world as part of Windows 8 non-deterministic LFH heap Valasek and Mandt (2012). However, randomization in heap has been discussed in a number of prior studies, and their randomization method differs from one to another. For example, Bhatkar et al. and Qin et al., respectively randomize the base address of the heap, as shown in Refs. Bhatkar et al. (2003); Qin et al. (2005). Additionally, there are others approaches that randomize heap chunk size during allocation phase Iyer et al. (2010); Kharbutli et al. (2006). Finally, other work focus on randomizing heap metadata Berger and Zorn (2006); Bhatkar et al. (2005); Novark and Berger (2010). Furthermore, Ref. randomizes the order of heap chunk allocation as well.

7. Conclusion

In this paper, we presented BadASLR: a set of peculiar cases where ASLR counter-intuitively aiding memory exploitation. ASLR is, without a doubt, one of the most successful and popular exploit mitigation technique well deployed in real-world. However, according to our study, there are four types of BadASLR in theory, and some of the cases were actually found in real-world exploitation. We do not argue to change/fix the

current randomization scheme as plus side of ASLR if significantly greater, but we hope our research can be a thought-provoking paper for advancing the completeness of knowledge.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was supported by the Sungshin Womens University Research Grant of 2021.

REFERENCES

Akritidis P. Cling: A memory allocator to mitigate dangling pointers. In: USENIX Security Symposium; 2010. p. 177–92.
Berger ED, Zorn BG. Diehard: probabilistic memory safety for unsafe languages, 41. ACM; 2006. p. 158–68.
Bhatkar S, DuVarney DC, Sekar R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits, 3; 2003. p. 105–20.
Bhatkar S, DuVarney DC, Sekar R. In: Usenix Security. Efficient techniques for comprehensive protection from memory error exploits; 2005.
Cve,2018-2018-5200. KMPlayer 4.2.2.15 and earlier have a Heap Based Buffer Overflow Vulnerability.
Caballero J, Grieco G, Marron M, Nappa A. 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. p. 133–43.
Coffman J, Kelly DM, Wellons CC, Gearhart AS. Rop gadget prevalence and survival under compiler-based binary diversification schemes. In: Proceedings of the 2016 ACM Workshop on Software PROtection; 2016. p. 15–26.
Davi L, Sadeghi A-R, Lehmann D, Monrose F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14); 2014. p. 401–16.
Davi LV, Dmitrienko A, Nürnberger S, Sadeghi A-R. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security; 2013. p. 299–310.
Follner A, Bartel A, Bodden E. Analyzing the gadgets. In: International Symposium on Engineering Secure Software and Systems. Springer; 2016. p. 155–72.
Shellphish„how2heap. Educational Heap Exploitation.
Hiser J, Nguyen-Tuong A, Co M, Hall M, Davidson JW. Ilr: Where'd my gadgets go?. In: 2012 IEEE Symposium on Security and Privacy. IEEE; 2012. p. 571–85.
Iyer V, Kanitkar A, Dasgupta P, Srinivasan R. Preventing overflow attacks by memory randomization. In: Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on. IEEE; 2010. p. 339–47.
King, soft. WPS Complete office suite with PDF editor.
Function, granular kernel address space layout randomization.
Kharbutli M, Jiang X, Solihin Y, Venkataramani G, Prvulovic M. Comprehensively and efficiently protecting the heap, 41. ACM; 2006. p. 207–18.
Kikuchi H, Arimizu T. On the vulnerability of ghost domain names. In: 2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. IEEE; 2014. p. 584–7.
Kil C, Jun J, Bookholt C, Xu J, Ning P. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC'06). IEEE; 2006. p. 339–48.
Lee B, Song C, Jang Y, Wang T, Kim T, Lu L, Lee W. Preventing use-after-free with dangling pointers nullification. Proceedings of the 2015 Internet Society Symposium on Network and Distributed Systems Security, 2015.
Li J, Wang Z, Jiang X, Grace M, Bahram S. Defeating return-oriented programming through gadget-less kernels. In: Proc. European Conference on Computer Systems; 2010. p. 195–208.
Microsoft,2016Cve-2016-0191. Edge allows remote attackers to execute arbitrary code.
Mortimer T. Removing rop gadgets from openbsd. Proc. of the AsiaBSDCon 2019:13–21.
Nagarakatte S, Zhao J, Martin MM, Zdancewic S. Cets: compiler enforced temporal safety for c, 45. ACM; 2010. p. 31–40.
Novark G, Berger ED. Dieharder: securing the heap. In: Proceedings of the 17th ACM conference on Computer and communications security. ACM; 2010. p. 573–84.
Qin F, Tucek J, Sundaresan J, Zhou Y. Rx: treating bugs as allergies—a safe method to survive software failures, 39. ACM; 2005. p. 235–48.
Seo J, Lee B, Kim SM, Shih M-W, Shin I, Han D, Kim T. Sgx-shield: Enabling Address Space Layout Randomization for Sgx Programs; 2017. NDSS
Shacham, Hovav. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proceedings of the 14th ACM conference on Computer and communications security 2007:552–61 In press.
Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi A-R. In: 2013 IEEE Symposium on Security and Privacy, IEEE. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization; 2013. 574–588
Serebryany K, Bruening D, Potapenko A, Vyukov D. Addresssanitizer: A fast address sanity checker. In: USENIX Annual Technical Conference; 2012. p. 309–18.
Stancill B, Snow KZ, Otterness N, Monrose F, Davi L, Sadeghi A-R. Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In: International Workshop on Recent Advances in Intrusion Detection. Springer; 2013. p. 62–81.
Valasek C, Mandt T. Windows 8 heap internals. Black Hat USA 2012.
Wartell R, Mohan V, Hamlen KW, Lin Z. Binary Stirring: Self-randomizing Instruction Addresses of Legacy X86 Binary Code. In: Proceedings of the 2012 ACM conference on Computer and communications security; 2012. p. 157–68.
Wressnegger C, Yamaguchi F, Maier A, Rieck K. Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security; 2016. p. 541–52.
Xu W, DuVarney DC, Sekar R. An efficient and backwards-compatible transformation to ensure memory safety of c programs. ACM SIGSOFT Software Engineering Notes 2004;29(6):117–26.
Younan, Y., 2015. Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers.

Daehee Jang received the B.S. degree in Computer Engineering from Hanyang University, South Korea, in 2012. He also received the M.S. degree in Information Security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014. He received Ph.D. of Information Security at KAIST in 2019; and he worked as postdoctoral researcher at Georgia Tech until 2020. He participated in various global hacking competitions (such as DEFCON CTF) and won several awards. He received a special prize from 2016 Korean government annual event for finding 0-day security vulnerabilities in many software products.