Received October 19, 2020, accepted October 28, 2020, date of publication November 5, 2020, date of current version November 18, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.3036118*

# HS-Pilot: Heap Security Evaluation Tool Model Based on Atomic Heap Interaction

**SUMIN CHAE, HONGJOO JIN, MOON CHAN PARK, AND DONG HOON LEE, (Member, IEEE)**
Graduate School of Information Security, Korea University, Seoul 02841, South Korea

Corresponding author: Dong Hoon Lee (donghlee@korea.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government Ministry of Science and ICT (MSIT) under Grant NRF-2017R1A2B3009643.

**ABSTRACT** To evaluate heap security, researchers have designed evaluation tools that automatically locate heap vulnerabilities. Most of these tools define heap interactions as heap misuses that are bugs, such as overflow in a target heap allocator, and verify whether each combination of heap interactions can be used as an exploit. However, this definition of heap interactions requires preliminary work by a user possessing evaluation tools and specialized knowledge—the user needs to manually do much work to find which heap misuses exist in the target heap allocator. In addition, because the existing heap misuses vary according to target heap allocators and versions, this preliminary work must be performed on each heap implementation. That is, the current definition of heap interaction cannot be generalized to all heap implementations. In this article, we propose a novel heap security evaluation model, called *Heap Security Pilot* (*HS-Pilot*), to overcome the preliminary work load and the dependency of heap misuse in heap implementation. In *HS-Pilot*, a heap interaction is newly defined as the modification of heap metadata, based on the idea that any heap misuse can be represented by a sequence of heap metadata, i.e. combination of heap interactions used by *HS-Pilot*. Consequently, the heap interactions in *HS-Pilot* can be applied to all heap implementations without specialized knowledge, and therefore, are more general than that in existing heap evaluation tools. Our evaluation shows that *HS-Pilot* can cover the analysis range of other evaluation tools, and is able to detect 14 known types of heap exploitation against heap allocator ptmalloc and all types of heap exploitation found by a state-of-the-art evaluation tool.

**INDEX TERMS** Computer security, memory defenses, software testing.

## I. INTRODUCTION

Most modern heap allocators—such as dlmalloc, ptmalloc and musl—use an inline metadata approach in which the metadata and the user data are placed in adjacent memory regions. The inline metadata approach is a consistently popular heap implementation method due to the advantages of cache and memory-saving features [8]. However, this design is vulnerable to metadata corruption attacks such as overflow. Heap allocators without security considerations can easily cause anomalies by tampering with the metadata. Currently, a number of heap vulnerabilities have been leveraged for malicious software activity [30].

Many studies on the heap allocator have been considered for security aspects [33–[35]. As its importance emerged, researchers also proposed a variety of evaluation tools for heap security [6]–[8]. Existing tools use heap misuses such as overflow, use-after-free, and double-free to verify whether the combinations of heap interactions containing the misuses can be used as heap exploitation primitives. Unfortunately, the tools based on heap misuses have two limitations: the need for preliminary work and the specificity of heap misuse.

First, in order to use existing evaluation tools, an evaluation tool user must first manually locate the existence of heap misuses on a target heap allocator before using the evaluation tool. This preliminary analysis requires expertise and user judgment to determine the appropriate parameters for heap misuse. Users using these evaluation tools find it difficult to acquire expertise and meet the challenge of being responsible

The associate editor coordinating the review of this manuscript and approving it for publication was Xiangxue Li.

for the wrong range of heap misuse. For example, when using overflow as a heap interaction in existing evaluation tools, the user should select a threshold to set how many bytes can be changed by overflow. If the threshold is not large enough, the evaluation tools are not able to find a possible heap exploitation. Conversely, if the threshold is too large, it may take the existing tools a long time to analyze many cases with long, overflowed data. Therefore, to set the threshold at a proper value, the user must have deep knowledge of exploitation techniques, making it difficult for some users to use these existing tools.

Second, the preliminary analysis described above is dependent on each implementation of heap allocators. More specifically, a heap misuse may occur or disappear on a new version even in a heap allocator. This means that it takes enormous effort to find heap misuses that are not explicit in existing tools. As an example of the first case, in glibc version 2.25, a new function called tcache was introduced to improve performance. However, a lack of security awareness about tcache causes a new heap misuse that did not occur in previous versions. Conversely, in regards to the second case, a double-free occurred in dlmalloc version 2.7.2, but this misuse no longer works on dlmalloc version 2.8.2 or later versions. Therefore, heap misuse cannot be generally defined for various heap implementations, and the preliminary work must be done as needed for each allocator and version [10].

In this article, we propose a new heap security evaluation model to solve the limitations of existing heap evaluation tools. In *HS-Pilot*, we newly define heap interaction as the modification of heap metadata. For the preliminary work for *HS-Pilot*, only the metadata structure in the target heap allocator is required, but it is not a burden to find the metadata information because it is publicly available from source code or paper. For this advantage, *HS-Pilot*, hardly affected by the difference of allocator implementations, can be generally applied to various implementations.

In addition, each heap misuse in existing tools can represent a combination of modifications of heap metadata. This modification on heap metadata is semantically the smallest behavior in a heap exploit, and therefore, can be considered *atomic heap interactions*. By using the *atomic heap interactions*, *HS-Pilot* can cover the analytical range of other evaluation tools. The main contributions of this article are summarized as follows:

1) We propose a new heap security evaluation model, *HS-Pilot*, which can be utilized without specialized knowledge and much preliminary work.
2) In *HS-Pilot*, we newly define *atomic heap interaction* based on heap metadata. The *atomic heap interactions* make it possible for *HS-Pilot* to be applied to various heap allocator implementations.
3) *HS-Pilot* can cover the analysis range of existing evaluation tools because heap interactions in existing tools can be represented combinations of *atomic heap interactions* used by *HS-Pilot*. Moreover, *HS-Pilot* can find some exploits that existing evaluation tools cannot.

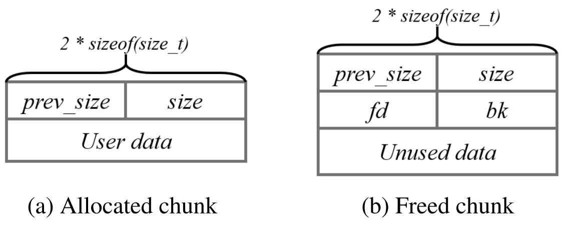

**FIGURE 1. Two types of chunk states in ptmalloc, in which the size corresponds to smallbin.**

4) We evaluate *HS-Pilot* on some allocators. In ptmalloc, *HS-Pilot* can find all 14 known heap exploitation techniques, as compared to 8 techniques found and 1 technique not found by *HEAPHOPPER*.

## II. BACKGROUND

Each application implements or selects an appropriate heap allocator. In modern embedded systems such as devices in IoT environments, there are not enough resources. In these cases, a lightweight C library such as musl [20] can be used, and the operation for dynamic memory allocation is also simplified. However, streamlined function can contain system faults through unsecure memory allocation. To prevent the situation of exploiting heap, heap allocator is evaluated by evaluating tools. In our approach, to analysis heap security, an user of an evaluation tool have to know the structure of the chunk that is basic unit of the heap allocator. This section introduces the design of heap allocator that use in inline metadata along with attacks that occur using heap allocator.

### A. DESIGN OF HEAP ALLOCATOR

There are various heap allocators to manage the heap of an application. These heap allocators use a similar structure of heap metadata to achieve a common goals that are to maximize compatibility, portability, locality, and error detection while minimizing time, space and anomalies [17]. We discuss the structure of metadata in ptmalloc, the most popular heap allocator. Ptmalloc is an allocator derived from dlmalloc [17], and is the base design of various allocators such as Quickfit [19] and musl. The GUN C library project, commonly known as glibc, also uses ptmalloc as the default heap allocator and as a core library for systems using Linux as the kernel [21].

Chunks are sorted by 8 bytes in a 32-bit architecture and each metadata fields are aligned by *sizeof(size_t)*. Ptmalloc maintains metadata corresponding to allocated or freed chunk as shown in Fig. 1 depending on the state of memory. The allocated chunk has two pieces of metadata: *prev_size* and *size*. In the *size*, the last three bits are not used due to the aligned size and then store three information. The first is *PREV_INUSE* bit that is the lowest bit of size. When the previous chunk is in use, the bit is set the value to 1 and if the previous chunk is freed, the bit is stored the size of the previous chunk in the *prev_size* and change the value to 0. The second is *IS_MMAPPED*, which indicates whether the chunk is allocated by the *mmap()* system call. Memory region created by calling system functions is managed differently

**TABLE 1. A description of the heap exploitation technique in terms of metadata tampering and the resulting exploitation type.**

| Name | Type of vulnerability | Description |
|---|---|---|
| fastbin_dup | Overlapping Allocation | Allocate a chunk that is not freed by corrupting freelist of fastbin |
| fastbin_dup_consolidate | Overlapping Allocation | Doubly allocate a chunk by simultaneously inserting in freelist of unsorted bin and fastbin |
| overlapping_chunks | Overlapping Allocation | Reallocate a chunk that is not freed by modifying size of freed chunk |
| overlapping_chunks2 | Overlapping Allocation | Reallocate a chunk that is not freed by modifying size of allocated chunk |
| poison_null_byte | Overlapping Allocation | Overlapping allocated chunk by using fake previous size and modifying size |
| fastbin_dup_into_stack | Non-Heap Allocation | Allocate fake chunk by modifying the forward pointer of freed chunk |
| house_of_force | Non-Heap Allocation | Allocate a chunk larger than the application can allocate by modifying the size of top chunk |
| house_of_lore | Non-Heap Allocation | Allocate fake chunk by passing unlink security check |
| house_of_spirit | Non-Heap Allocation | Allocate fake chunk in freelist by freeing fake chunk |
| house_of_einherjar | Non-Heap Allocation | Allocate fake chunk by modifying previous size and size of allocated chunk |
| unsortedbin_into_stack | Non-Heap Allocation | Overwrite target address by returning fake chunk |
| large_bin_attack | Arbitrary Write | Overwrite target address by abusing unlink macro routine |
| unsafe_unlink | Arbitrary Write | Write the desired value using unlink macro of fake chunk |
| unsortedbin_attack | Arbitrary Write | Overwrite target address by modifying backward pointer of freed chunk |

from existing chunks. The last is the *NON_MAIN_ARENA* bit. Ptmalloc, which supports the thread function, manages the heap memory area per thread. The flag bit is activated when an allocated chunk is created to other thread rather than the main thread.

The freed chunks are managed by bins, which are structures of a freed chunk list for each size. The freed chunk has four pieces of metadata in smallbin: *prev_size*, *size*, *fd(forward pointer)* and *bk(backward pointer)*. The reason that freed chunks have more metadata than allocated chunks is because they contain information to efficiently reuse memory. The first two metadata fields play the same role as in an allocated chunk, and the latter two are used to maintain freed chunk list of the same size or within a specified range. To add metadata fields such as *fd* and *bk*, unused memory is used as metadata fields without additional memory allocation.

The heap creation for an application uses the top chunk design for a small memory footprint. It contains information about the available size of the heap. If there is no suitable sized chunk for allocation, the top chunk returns the new chunk of the size to be created, keeping the rest as top chunks. If there is no space, the usable heap area is increased through *sbrk* or *mmap* system call.

There is a technique that modifies these top chunk as well as one that corrupts the chunk metadata. Therefore, top chunk should be considered as a element of security evaluation. It is difficult to detect technique such as *house_of_force* because address of top chunk change fluidly. In order to tracing top chunk, *HS-Pilot* use hook function that can capture address of top chunk in runtime.

### B. HEAP EXPLOITATION TECHNIQUES

There are various techniques for exploiting the heap shown as TABLE 1, and can be broadly classified into two types: The first is passive heap exploitation, which is an attack method that uses only valid function of the heap without modifying any metadata, such as malloc and free [13]. The passive attack is executed by incorrect heap implementations, but many modern heap implementations now defend against these attacks due to a fund of knowledge. The other type is active heap exploitation, which include most heap exploitation techniques. An active attack leads to malfunction of heap by manipulating metadata and bypassing security-checking routine applied in heap implementation. Since security checking routines are not perfect, heap evaluation tools can help by making up for the missing part for improving heap security.

The type of a heap exploitation techniques essentially has three prongs: Overlapping Allocation, which overwrites all or part of a chunk already allocated through the heap API; Non-heap Allocation, which allocates a chunk through malloc to non-heap areas(e.g., stack); and Arbitrary Write, which overwrites a limited or arbitrary value in a desired address.

## III. RELATED WORK

In the past, heap exploitation techniques were discovered by attackers manually analyzing the heap implementation. Attacks related to heap did not get the attention of attackers, because it took a lot of effort to succeed. However, automatic exploit generation tools for heap [9], [31], [32] have been recently made so that attackers can easily create exploitation. Attacks that use vulnerabilities of heap allocator are simply reproduced if the memory in the heap is controlled by attackers. It means that the same vulnerability exist in applications that use the corresponding heap allocator.

Applications can also be attacked without a deep understanding of applications, because the heap vulnerabilities are application independent. Due to application's independence, heap vulnerabilities are more critical to security than vulnerabilities that depend on the implementation of a particular application. Heap exploitation have been studied extensively [16], [26] in the field of software security, and many researcher have attempted to develop framework, using a variety of innovative automation tools (e.g., model checkers, fuzzing and symbolic execution) [9].

*HEAPHOPPER* [7] is the first automatic evaluation tool thorough model checking and symbolic execution to evaluate the security of a heap implementation. By detecting the heap exploitation primitive, *HEAPHOPPER* helps to

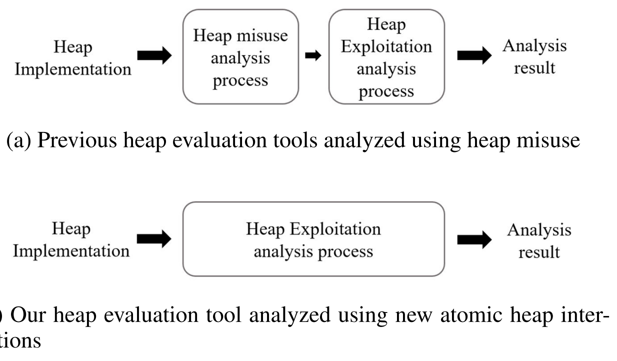
Heap Implementation
Heap misuse analysis process
Heap Exploitation analysis process
Analysis result
(a) Previous heap evaluation tools analyzed using heap misuse
Heap Implementation
Heap Exploitation analysis process
Analysis result
(b) Our heap evaluation tool analyzed using new atomic heap interactions

**FIGURE 2.** **Comparison of our mechanism with previous study.**

identify implementation problems in heap implementation. The framework defines heap interaction based on heap misuse and analyzes it using the symbolic execution engine *angr*. In similar way, *ARCHEAP* [8] is the latest heap allocator evaluation tool through fuzzing. It defines similar heap interaction like *HEAPHOPPER*. However, rather than checking all combinations of heap interactions, *ARCHEAP* devises a way to quickly estimate the possibility of exploitation primitive and generate combinations of heap interactions. It uses the american fuzzy lop (AFL) [28] fuzzer to generate the input value of the heap interaction, and the shadow memory is designed for real-time analysis.

Heap implementations can be analyzed using tools, of which the most common are fuzzing and symbolic execution. Fuzzing helps to find exploitable input by generating random values, and an evaluation tool enables efficient exploitation detection. Fuzzing generates random value depending on criteria and done not consider semantic of heap implementation. Therefore, fuzzing does not guarantee security of the heap allocator even if the exploit has not been detected. In *HS-Pilot*, a combination of heap interactions is analyzed using symbolic execution, which has wide analysis coverage because symbolic execution generate executable path that reflects semantics of heap implementation.

In Fig. 2, our analytical mechanism is shown compared to previous studies. Previous papers discovered heap exploitation techniques [6]–[8] by constructing heap interactions based on heap misuse. However, because security routines are different between allocators and versions, it is necessary to determine the heap misuse that may occur and generate a coordinated combinations of heap interaction. In this article, we define atomic heap interactions through information on the metadata structure of heap. As a result, our tool does not require additional analysis unlike previous research.

## IV. DESIGN

### A. ADVERSARY MODEL

This section presents the capabilities of an attacker. An attacker aims to identify vulnerabilities in target heap allocator, and evaluation tool has the same goal as the adversary model. Therefore, we give the evaluation tool the same abilities as the actual attackers. By modeling powerful attacker capabilities, we analysis heap implementation similarly with a real environment and effort to identify all known heap exploitation techniques. Modern protection techniques such as address space layout randomization (ASLR) [25] make it difficult to found heap exploitation techniques, but these defense do not essentially improve the security of heap implementation. As bypassing protection techniques is out of scope in our research, we assume that there is no protection techniques except for the security checking routine implemented in heap implementation.

Because heap APIs represent how they work in real programs based on heap implementations, evaluation tools should be able to run valid heap APIs. According to standard documentation [22], functions(heap APIs) that manage heap memory are malloc, free, calloc, realloc, memalign, etc. However, an increase in the number of heap interactions can result in a large overhead of the evaluation tool. So we replaced alternative heap APIs to prevent this. Other allocation APIs (excluding malloc and free) use malloc or free inside. In addition, they perform additional actions such as resizing or copying user data. These additional actions are meaningless from a metadata perspective, which means they can be replaced by malloc and free. For this reason, the previous papers also evaluated heap allocators using malloc and free.

To discover potential heap exploitation, the evaluation tool changes heap metadata, and then bypasses security checking routines. The heap implementation design is specified in the annotation or white paper, from which we can also easily obtain information about the metadata. The attacker can modify metadata as a field of chunk or exploit indirect metadata from the target heap allocator, such as the top chunk. Similarly, *HS-Pilot* analyzes corrupt metadata through symbolic execution. Indirect metadata is not metadata of each chunk, but can indirectly affect the management of all chunks. This indirect metadata can be obtained with the same behavior used in previous heap exploitation techniques, such as modifying the top chunk.

### B. HS-PILOT

To evaluate the security of the heap implementation, we introduce *HS-Pilot*, an automated approach to generating heap exploits through heap metadata corruption. *HS-Pilot* controls the heap in the same way as the adversary model that presented earlier in section 4.1. The operation process of *HS-Pilot* is shown in Fig. 3. To generate combinations of heap interactions, we use *configuration file* and *code generator*.

*Configuration file* consists of information needed for analysis such as the location of a target heap allocator, set of heap interactions, bound and temporal/spatial constraints. The location of heap allocator is the directory path of target heap allocator, and the set of heap interactions are valid heap interactions extracted against the target heap allocator based on section 4.3. Since *HS-Pilot* analyzes combinations of heap interactions, we need to specify a bound which means the maximum combination length for finality of analysis. Also,

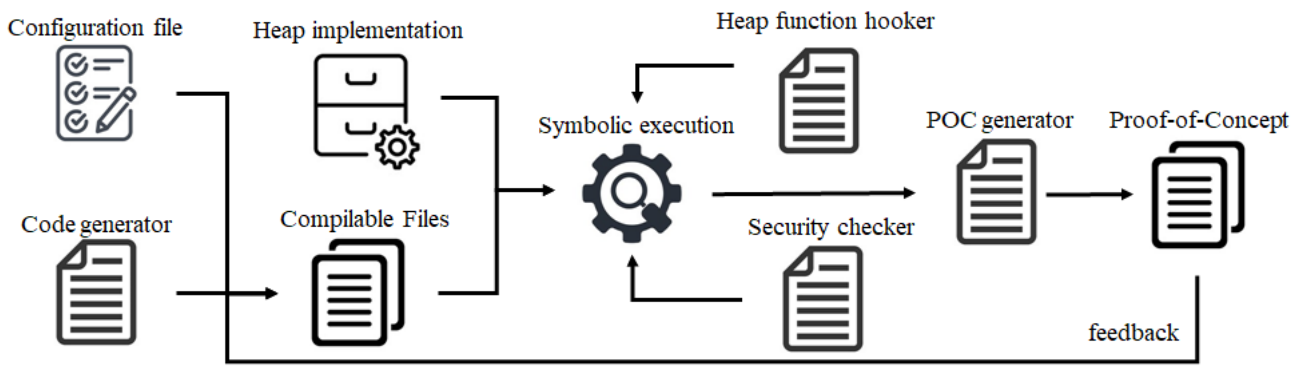

**FIGURE 3.** Overview of *HS-Pilot*.

we have to configure temporal/spatial constraints because of limitation on system resource.

**Algorithm 1** Generating Compilable Files With a Heap Interaction (HI)

```
Input  : $File_{Config}$
Output: $CompilableFiles$

$D \leftarrow$ GetDepth($File_{Config}$)
$HI_{Pre} \leftarrow$ GetPreHI($File_{Config}$)
$HI_{Set} \leftarrow$ GetHI($File_{Config}$)
$Size_{malloc} \leftarrow$ GetSize($File_{Config}$)

for $D_{Curr} \leftarrow 2$ to $D$ do
    $HI_{Combs} \leftarrow$ DoCombination($HI_{Set}, D_{Curr}, HI_{Pre}$)
    $HI_{Conc-Combs} \leftarrow$ DoAppendIdx($HI_{Combs}$)
    $Vulnerable_{Combs} \leftarrow$ GetVulnerableCombs($File_{Config}$)
    for A combination $HI_{Single-Comb} \in HI_{Conc-Combs}$ do
        if $HI_{Single-Comb} \in Vulnerable_{Combs}$ then
            Remove($HI_{Single-Comb}$)
        else
            MakeFile($HI_{Single-Comb}$,$Size_{malloc}$)
        end
    end
end
```

*Code generator* makes a compilable code based on combination of heap interactions. This process can be represented by pseudo-code, as shown in Algorithm 1. To minimized unnecessary analysis, a predecessor can be set for each heap interaction. For example, to modify the *forward pointer* of a chunk, a chunk is required. In other words, it is invalid that modify *forward pointer* before allocating chunk. Consequently, We set predecessors for heap interactions by using relationships between metadatas. Setting the predecessors has the advantage of optimization for analysis phase, and the accuracy of *HS-Pilot* is not affected, even if predecessors are not set.

For analysis only through metadata modification (to avoid creating other unnecessary paths), we perform heap interaction specification. Therefore, each heap interaction have to specify chunk that perform that action in the combinations of heap interactions. To differentiate the chunks used for analysis, Allocated chunks maintain each unique index in order. The concretization process uses indexes to specify which chunks to modify through heap interaction. Then, *HS-Pilot* creates all possible combination by using concrete heap interactions. Since modifying with metadata that has already been tampered is unnecessary, combinations with duplicate heap interactions are eliminated beforehand. Because it is pointless to tamper with metadata in the same chunk, theses combinations are eliminated in advance. In addition, the Proof of Concept (PoC) code is stored to reflect the previous analysis result, and the newly created combinations are compared with the base combination of the PoC code. If the generated combination contains a combination based on PoC code, *HS-Pilot* removes the combination.

After compiling generated combinations, the executable files are analyzed with the *angr* framework [24], a binary-based symbolic execution engine. Source code-based analysis is difficult to reflect information generated and changed at runtime. Because it is difficult to find the corruption of memory management, we use a binary-based symbolic execution engine. Symbolic execution restricts the value of a symbolic variable using constraints on the basis of control flow. For our analysis, metadata that is modulated through heap interaction is set as a symbolic variable, and the information necessary for runtime analysis, such as address of top chunk, is obtained through defined hook function. To efficiently detect heap exploitation primitive, we set a read/write strategies for symbolic variables. According to existing heap exploitation, analysis is facilitated via a value based on a target address, allocated chunk address or fake chunk address. Therefore we use these specified address for the read/write strategies.

During the analysis, we need criteria to detect the heap allocators malfunctioning. So we use *security checker* to detect security violations of target heap allocator. We define three security violations for heap.

1) Overlapping Allocation: When allocating a chunk, it means that the previously allocated chunk and the chunk to be allocated through the malloc function become overlapped. The *security checker* verifies the existence of coincidence between the allocated malloc list and the chunk to be allocated using address and size.
2) Non-Heap Allocation: It means that the chunk to be allocated through the malloc function is created outside the heap section. The malloc function allocates unused memory in the heap, but exploiting heap allocator can return non-heap memory. For example, because of a memory locality, the address of the fake chunk that is not in heap is allocated, and this situation can occur when freeing a fake chunk and allocating the same size memory.
3) Arbitrary Write: This means to write an arbitrary or limited value at the target address. Malloc and free contain code that writes memory inside the function to manage metadata. By exploiting mechanism about metadata change, we can modify the value of the target address. The value of the target address can only be modified to a restricted value if the mechanism has constraints. If there is no constraint, the value can be modified to any value. We observe whether the value of the target address has changed after calling malloc and free to detect Arbitrary Write.

If a state that is a security violation is reached during analysis, *HS-Pilot* obtains a value that satisfies the constraints of the symbolic variable for the violated state, and generates PoC code based on the value. To reduce the duplication of the analysis, we gradually increase the length of the combination and reflect the results of the previous analysis.

## C. ATOMIC HEAP INTERACTION

A heap interaction refers to what can be done with an evaluation tool to actually verify target heap allocator. In previous researches, heap interactions were defined based on heap misuses. Because it would be pointless to evaluate security of heap implementation using a non-existent heap misuse, it is necessary to check existence of heap misuse in target heap allocator before analysis. Checking step requires expert knowledge and effort. Also, different versions of the heap misuse occur even with the same heap allocator.

In this article, we define *atomic heap interactions* through metadata of heap. Each metadata represents the smallest unit containing one piece of information for managing the heap and is no longer semantically divisible. Therefore, we call an heap interaction based on metadata as *atomic heap interactions*. Identifying the metadata structure of a heap implementation is a well-known piece of information without analysis, and is easier than figuring out the heap use of the target heap allocator. This definition overcomes the limitation of existing research that require expertise. In our research, *HS-Pilot* accesses different metadata via offsets based on the chunk address. Since the size of each metadata is fixed, we do not need setting a misuses of threshold such as a overflow size. *Atomic heap interactions* have four types.

- **Dynamic memory related heap interaction** - This means a valid API to be used on the heap such as malloc and free, and these functions dynamically manage memory. Malloc receives input parameter that is the size to be allocated and returns the address of the allocated chunk. Information about allocated memory is then maintained as a global variable, and the size to be allocated can be specified through a *configuration file*. Free receives input parameter that consists of an address to be deallocated with no return value. The argument of the free function can include the address value assigned through malloc function and fake chunk.
- **Malloc metadata-based heap interaction** - It is a type of heap interaction that manipulates the metadata of allocated chunks. The allocated chunk has metadatas such as a chunk size, flag bits and previous chunk size that is the size of the chunk adjacent to the current chunk.
- **Free metadata-based heap interaction** - This heap interaction manipulates freed chunk metadata. In general, freed chunks are managed by size through a linked list.
  for efficient use of allocated chunk metadata and memory. If the allocated chunk is fake free, free metadata-based heap interaction can be used for the chunk that is not free. In other words, chunks allocated through the heap API can all use heap interactions about metadata used in malloc and free, regardless of state.
- **Indirectly affecting heap interaction** - This heap interaction indirectly affects the heap and does not belong to the above three interaction types. This type includes modifications to metadata for heap management, such as top chunk, or memory allocation for connections with other areas, such as fake chunks. This heap interaction can be identified through actions used in known heap exploitation techniques.

*HS-Pilot* uses an defined *atomic heap interactions* through above criteria. Since the heap area is initialized after the first malloc call in the program, we use dummy chunks to perform analysis in the initialized heap state (allocation and deallocation). So we can use initialized heap state information such as top chunk. Also, fake chunks that are created outside of the heap are not affected by sequence of heap interactions, so *code generator* initially places fake chunks to create combinations that minimize duplication.

## D. HEAP HOOK FUNCTION

*HS-Pilot* analyze using a lot of information that is generated at runtime. Because the address to be read or written differs for each combination of heap interactions, information related to the heap have to be collected at runtime. In addition, it should be maintained that fake chunk address and information about the top chunk address. However, since runtime information is not shown in source code, we use hook functions to obtain runtime information. *HS-Pilot* uses four hook functions.

- **Malloc hook function** - Heap API is difficult to analysis because of complex mechanism such as memory consolidation. *HS-Pilot* uses values that are returned directly through malloc function, so there is no need to consider complex mechanisms. Malloc hook function stores the size and address of the chunk in a malloc chunk list when malloc function is called from binary file. *HS-Pilot* is able to verify that keeping security policy, such as checking whether it is allocated inside the heap through the stored address.
- **Free hook function** - Similar to the Malloc function, free hook function occurs when the free function is called from a binary file. *HS-Pilot* remove an address that received input of free function in malloc chunk list, and insert the address in free chunk list. If the address is not in malloc chunk list, insert only free chunk list. This situation is fake free if the address is not in free chunk list. Contrastively, if the address is in free chunk list, it is double free.
- **Fake chunk hook function** - Fake chunk is data that maintains a similar structure to chunk in non-heap section. The goal of fake chunk is to make target heap allocator identify as legitimate chunk. The fake chunk address is what must be known for heap interaction input to create a valid exploitation technique. When analyzing this information, note that the address of the fake chunk stored on the stack may be different from the actual runtime memory location. For this reason, when we discovered a heap exploitation technique, we used the hook function to preserve address information for fake chunks and reanalyze the information.
- **Top chunk hook function** - Position of top chunk depends on chunks that are presently allocated, and *size* metadata field of top chunk keep how much memory can newly be allocated in the heap. The goal of exploitation is to allocate more than the size that can be allocated in the heap by modifying the *size* field. Top chunk hook function find address of the top chunk. In ptmalloc, *HS-Pilot* finds the address of the top chunk by referring to the *main_arena* structure that manages the heap area inside the libc function.

*HS-Pilot* uses four defined hook functions to obtain runtime information. Malloc and Free hook function are occurred when malloc function and free function are called, and fake chunk and top chunk hook function are hooked by defining special function.

## V. IMPLEMENTATION

In this article, we implemented the prototype model, *HS-Pilot*, to analyze ptmalloc, the allocator of glibc version 2.23. In *configuration file*, we use YAML format that is a data serialization language. The process of code generation and analysis is written by python version 3.5.4 and with a totals 1500 line of code. We generate code using a file format that is compile with GNU compiler collection(GCC) on linux, and analyze it with version 8.19.4.5 of the *angr* framework, which is a binary symbolic execution engine.

```
size_t = 8

class Topchunkhook(angr.SimProcedure):
  def run(self, addr = None,
     var_dict_top_chunk_addr = None,
     var_dict_top_chunk_size = None):
    main_arena_top = addr + 88
    top_chunk_addr = self.state.mem[main_arena_top
     ].uint64_t.resolved
    top_chunk_index = self.state.heap_analysis.
     top_chunk_idx

    self.state.heap_analysis.top_chunk_idx += 1
    self.state.memory.store(
     var_dict_top_chunk_addr[top_chunk_index],
     top_chunk_addr, size_t, endness=archinfo.
     Endness.LE)
    self.state.memory.store(top_chunk + size_t,
     self.state.mem[var_dict_top_chunk_size[
     top_chunk_index]].uint64_t.resolved,
    size_t, endness = archinfo.Endness.LE)

    return


class FakeChunkAddr(angr.SimProcedure):
  def run(self, addr, var_dict_fake_addr = None):
    fake_chunk_index = self.state.heap_analysis.
     fake_chunk_idx

    self.state.heap_analysis.fake_chunk_idx += 1
    self.state.memory.store(var_dict_fake_addr[
     fake_chunk_index], addr, size_t, endness =
     archinfo.Endness.LE)
    self.state.heap_analysis.fake_addr.append(self
     .state.mem[var_dict_fake_addr[fake_chunk_index
     ]].uint64_t.concrete)

    return

```

**Listing. 1. Source code of hook function to obtain runtime information.**

For efficient analysis, *angr* provides an option called the plug-in function, which is simply simulated instead of a real function. Plug-in functions that are related to malloc and free are also implemented. However, we have to analysis real implemented functions so removed this option. If *HS-Pilot* encounters invalid path that lead to fail at runtime such as abort function, our tool halt analysis. Symbolic Execution performs objective analysis, and increases the reliability of the tool by creating an automated input without user intervention of the evaluation tool. We can get this advantage through using symbolic execution.

In *angr*, a global variable address can be caught through symbols of binary file. However, no symbols are generated for a local variables in the stack. In addition, several information is generated after heap section is initialized. For these information, *HS-Pilot* perform defined hook function. Listing 1 show source code of hook function to obtain address

**TABLE 2.** Description of combination for known heap vulnerability techniques generated by the code generator.

| Name | Combination of Atomic Heap Interactions |
|---|---|
| fastbin_dup | malloc→malloc→malloc→free→free→free→malloc→malloc→malloc |
| fastbin_dup_consolidate | malloc→malloc→free→malloc→free→malloc→malloc |
| fastbin_dup_into_stack | fake_chunk_out→malloc→malloc→malloc→free→free→free→malloc→malloc→fwd→malloc→malloc |
| house_of_force | malloc→top_chunk→malloc→malloc |
| house_of_lore | fake_chunk_out→fake_chunk_out→malloc→malloc→free→malloc→bck→malloc→malloc |
| house_of_spirit | fake_chunk_out→fake_chunk_out→malloc→free→malloc |
| house_of_einherjar | fake_chunk_out→malloc→malloc→prev_chunk_size→chunk_size→free→malloc |
| largebin_attack | malloc→malloc→malloc→malloc→malloc→malloc→free→free→ malloc→free→chunk_size→bck→malloc |
| overlapping_chunks | malloc→malloc→malloc→free→chunk_size→malloc |
| overlapping_chunks2 | malloc→malloc→malloc→malloc→malloc→free→chunk_size→free→malloc |
| poison_null_byte | malloc→malloc→malloc→free→chunk_size→prev_fake_chunk_size→malloc→malloc→free→free→malloc |
| unsafe_unlink | malloc→malloc→fake_chunk_in→prev_chunk_size→chunk_size→free |
| unsortedbin_attack | malloc→malloc→free→bck→malloc |
| unsortedbin_into_stack | fake_chunk_out→malloc→malloc→free→chunk_size→bck→malloc |

**TABLE 3.** Analysis results of 14 heap exploitation techniques detected using HS-Pilot.

| Name | Type of Heap Interaction | malloc size | Analysis time | Depth |
|---|---|---|---|---|
| fastbin_dup | [ malloc, free ] | [ 0x8 ] | 5.15 | 9 |
| fastbin_dup_consolidate | [ malloc, free ] | [ 0x8, 0x500 ] | 4.92 | 7 |
| fastbin_dup_into_stack | [ fake_chunk_out, malloc, free, fwd ] | [ 0x8 ] | 8.79 | 12 |
| house_of_force | [ malloc, top_chunk ] | [ 0x8, 0x100, 0x7fffffbf8840 ] | 7.62 | 4 |
| house_of_lore | [ fake_chunk_out, malloc, free, bck ] | [ 0x100, 0x500 ] | 10.44 | 9 |
| house_of_spirit | [ fake_chunk_out, malloc, free ] | [ 0x8, 0x100 ] | 7.71 | 5 |
| house_of_einherjar | [ fake_chunk_out, malloc, free, prev_chunk_size, chunk_size ] | [ 0x8, 0x100, 0x200 ] | 50.47 | 7 |
| largebin_attack | [ malloc, free, chunk_size, bck ] | [ 0x8, 0x100, 0x320, 0x500 ] | 12.57 | 13 |
| overlapping_chunks | [ malloc, free, chunk_size ] | [ 0x100, 0x500 ] | 4.36 | 6 |
| overlapping_chunks2 | [ malloc, free, chunk_size ] | [ 0x100, 0x200 ] | 14.40 | 9 |
| poison_null_byte | [ malloc, free, chunk_size, prev_fake_chunk_size ] | [ 0x100, 0x500 ] | 10.09 | 11 |
| unsafe_unlink | [ malloc, free, fake_chunk_in, prev_chunk_size, chunk_size ] | [ 0x100 ] | 54.75 | 6 |
| unsortedbin_attack | [ malloc, free, bck ] | [ 256 ] | 4.63 | 5 |
| unsortedbin_into_stack | [ fake_chunk_out, malloc, free, chunk_size, bck ] | [ 256 ] | 220.20 | 7 |

of top chunk and fake chunk. When the top chunk is created, the address of the top chunk is held in a structure called *main_arena* which is a specific value apart from libc. We use the top field of this structure to obtain the top chunk address at runtime and store the address by *size_t* bytes. Thus we are able to statically locate the address of the top chunk in the PoC code, and modify the size information of top chunk. A fake chunk can be created in the stack area or inside the chunk created with the malloc function. Similar the top chunk hook function, the fake chunk hook function is used to find the chunk address.

## VI. EVALUATION

The goal of *HS-Pilot* is to verify security about heap allocators. We evaluated the detection performance of against 14 known heap exploitation techniques in ptmalloc. In addition, we compared *HS-Pilot* with the *HEAPHOPPER*, a state-of-the-art heap evaluation tool. In this section, we discuss the following questions.

1) Can *atomic heap interactions* (newly defined heap interactions) generate combination for known heap exploitation techniques?
2) Can heap exploitation techniques be detected through the generated combinations?
3) How efficient is *HS-Pilot* compared to other state-of-the-art frameworks?

First, it is essential to verify that the *code generator* is able to generate a combination for known heap exploitation techniques. This process is equivalent to ensuring that *atomic heap interactions* are well defined, as an incorrectly defined heap interactions are not able to reproduce currently known heap exploitation techniques. TABLE 2 shows the combinations for the 14 heap exploitation techniques as described in section 2.2. The metadata that is used in largebin, such as *fd_nextsize* or *bk_nextsize*, can be modified through a fake chunk created inside target chunks. Because fake chunk triggers the constraints of the numerous symbolic variables, *HS-Pilot* has to resolve state explosion in symbolic execution. To prevent this situation, we limited number of possible uses of fake chunks.

Second, it should be checked whether the used evaluation tool properly detects heap exploitation. We describe our analysis using *atomic heap interactions* for known heap exploitation techniques as shown TABLE 3. Malloc size is candidate of parameter that is used in malloc function, and depth signifies bound that is length of the combination. Analysis time means the time that is required to analyze through *HS-Pilot*

**TABLE 4.** Table comparing ***HS-Pilot*** with ***HEAPHOPPER*** against 8 heap exploitation techniques.

| Tool / Technique | HS-Pilot | | HEAPHOPPER | |
|---|---|---|---|---|
| | All path | First vuln | All path | First vuln |
| **fastbin_dup_into_stack** | 13.71 | 8.79 | 4.64 | 3.63 |
| **house_of_lore** | 16.21 | 10.44 | 68.87 | 36.73 |
| **house_of_spirit** | 9.96 | 7.71 | 4103.47 | 6.50 |
| **overlapping_chunks** | 96.60 | 4.36 | 1552.5 | 36.68 |
| **unsortedbin_attack** | 4.90 | 4.27 | 2.98 | 2.60 |
| **unsafe_unlink** | 193.67 | 54.75 | 235.85 | 145.01 |
| **house_of_einherjar** | 134.38 | 50.47 | 1469.01 | 136.91 |
| **poison_null_byte** | 21.98 | 10.09 | 1523.83 | 78.26 |
| **Average Time** | ***61.42*** | ***18.86*** | ***1120.15*** | ***55.79*** |

when the corresponding combination of heap interactions is given.

Several types of vulnerabilities can occur in one combination of heap interactions, and *HS-Pilot* detects the type of vulnerability first discovered. For example, unsafe unlink is a heap exploitation technique that writes unrestricted values to a desired address. The process is as follows. We write $Addr_{object}$ as the address of the object. When we want to highlight the status of a given chunk, we denote $CH^A$ or $CH^F$ and metadata for a specific chunk is indicated as $metadata_{CH}$.

1) Allocate two chunks, $CH_1^A$ and $CH_2^A$, with the size of $0 \times 100$.
2) Store $Addr_{CH_1}$ and $Addr_{CH_2}$ in a global buffer $G$.
3) Create a fake chunk $FCH$ inside $CH_1$, where $prev\_size_{FCH}$ and $size_{FCH}$ are equal to 0. $fd_{FCH}$ and $bk_{FCH}$ are equal to $Addr_G - C_1$ and $Addr_G - C_2$, respectively, where $C_1$ and $C_2$ are constants.
4) Modify $prev\_size_{CH_2}$ and $size_{CH_2}$ to make $CH_1^A$ look like the fake freed chunk $CH_1^F$.
5) Free $CH_2^A$. After that, $CH_2^F$ and $CH_1^A$ are merged into one freed chunk. As a result, $Addr_{CH_1}$ in $G$ is overwritten.
6) Put the $Addr_{target}$ in $Addr_{CH_1^A+C_1}$. $Addr_{target}$ is stored in $G$ due to the manipulated $CH_1$. In this case, *target* is the object we want to tamper with.
7) Change the value of $Addr_{CH_1}$ to a desired value. Then, we can set the desired value in the target address.

The above heap exploitation technique is triggered by a change in the value of the global buffer through the unlink macro by freeing the second chunk. This means that this is an arbitrary write that writes a limited value at an arbitrary address, and *HS-Pilot* detects changes in the global buffer through a *security checker* in step5. The final goal of unsafe unlink can be achieved through the subsequent process.

Finally, the difference between *HS-Pilot* and existing evaluation tool is explained through comparison. We compares *HS-Pilot* with *HEAPHOPPER* as shown TABLE 4. We test 8 exploitation techniques that are detectable by *HEAPHOPPER*. In *HS-Pilot*, the length of combination is long because of using small units. Through a read/write strategy during symbolic execution, the check is performed on a specific address rather than all addresses, and the path of symbolic execution is constructed to mitigate the state explosion problem of symbolic execution. Heap misuse-based interactions result in a number of symbolic variables being allocated because target memory is not clear. In other words, many symbolic variables act as overhead of the evaluation tool, increasing analysis time. *HS-Pilot* uses a fixed length target memory to mitigate the state explosion in a symbolic extension. This can be seen in our comparison experiment between *HS-Pilot* using new atomic heap interactions and *HEAPHOPPER* using traditional heap misuses. *house_of_spirit* is a typical example of this.

For the finiteness of code execution, we set the tunable threshold. The temporal threshold was 1,500 seconds and spatial threshold was 8GB in used evaluation tools. *HS-Pilot* was able to analyze eight heap exploitation techniques within the threshold. Since we are not able to predict a point of vulnerability, we need to analyze each combination until the entire path is reached, not when you first find the vulnerability. To do this, we measured the time to detect the first vulnerability and the time to detect the entire path. The analysis was performed three times, and each value represents an average value. Because *HS-Pilot* has fewer symbolic variables than existing tools based on symbolic execution, *HS-Pilot* saves 67% on average when finding the first vulnerability and 95% when checking all paths.

In *HS-Pilot*, an address of a stack is specified. It mean that the address of the stack can be analyzed via read/write strategies by granting access to memory near the fake chunk. Also, by setting the memory layout, we are able to determine the gap between the heap and the stack. By using this information, we also found the specific malloc value of the *house_of_force* technique and subsequently generated a combination, enabling us to allocate the stack area address using the malloc function.

Additionally, we analyzed dlmalloc, which is identical to the chunk metadata of ptmalloc. Since it has the same metadata structure, *atomic heap interactions* used by ptmalloc can be used intact. However, even though the structure of the metadata is the same, we have to analyze dlmalloc because possible vulnerabilities differ according to internal implementation. We found that version 2.7.2 of dlmalloc is yielded the same vulnerabilities as ptmalloc, which was a highly probably outcome, as ptmalloc is based on dlmalloc 2.7.0. In order to set the top chunk to execute the *house_of_force* technique, ptmalloc stores information from the top chunk in a structure called *main_arena*, and dlmalloc 2.7.2 is stored in the *av_* structure. Since the newer dlmalloc 2.8.6 is not able to do double free, it is deemed safe against the fastbin heap exploitation technique. Also, version 2.8.6 of dlmalloc uses the *_gm_* structure to maintain information about the top chunk of the heap.

Finally, we analyzed musl 1.1.9 version. musl is similar in design with dlmalloc, so we were able to analyze it using atomic heap interaction without any significant burden. However, we checked the top of the heap with the *mal.brk* value without keeping the top chunk in the musl design. As a result, we found that modifying the top chunk resulted in an invalid interaction with musl and that the *house_of_force* technique could not be exploited. Similar to dlmalloc 2.8.6 version, we were able to apply double free defense techniques to show the invalidity of the previously known fast bin attack technique. In addition, the analysis using *HS-Pilot* showed that heap exploit techniques related to non-heap allocation for the stack were not valid. This is because there is a routine to check if the address is in the stack when allocating the chunk address in musl. Heap allocators are constantly developing to improve security, and existing manual tasks can be automated through heap evaluation tools. Before heap allocators are released, *HS-Pilot* can improve security by using our heap evaluation tool.

## VII. DISCUSSION

There are various vulnerabilities in heap implementations that cause heap corruption, but they must involve heap API use or heap metadata tampering. Use-after-free leads to information leakage about metadata and user data. In order for metadata information leakage to trigger heap corruption, a write action must occur on the metadata. *HS-Pilot* analyzes heap allocators based on metadata writes, so it is possible to recreate the manipulation of information leaked after use-after-free occurs. In addition, *HS-Pilot* considers both the memory area shared by user data and metadata itself as metadata. Because of this, user data through use-after-free is independent of metadata and does not compromise the integrity of heap implementations. The same goes for an uninitialized read and a buffer overread. Metadata management is done in the heap API, and information leakage through a memory read alone does not cause unintended behavior in the heap implementation.

## VIII. CONCLUSION

In this article, we proposed a new heap security evaluation tool *HS-Pilot* that uses atomic heap interactions. We verified a heap allocator without special knowledge about heap misuses. The number of combinations increases due to refined heap interaction, but this is alleviated through feedback. In addition, *HS-Pilot* also showed that the analysis time for each combination improved over existing evaluation tools that use symbolic execution. Able to detect all 14 known heap exploit techniques for ptmalloc, *HS-Pilot* is proven to have a wider analysis range than *HEAPHOPPER*.

## REFERENCES

[1] *TIOBE Index for May 2020*. Accessed: May 12, 2020. [Online]. Available: https://www.tiobe.com/tiobe-index/

[2] E. Ruchko, "The six best-paid IoT programming languages," Informa, London, U.K., Nov. 2017. [Online]. Available: https://www.iotworldtoday.com/2017/11/08/six-best-paid-iot-programming-languages/2/

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th Conf. USENIX Secur. Symp.*, San Antonio, TX, USA, vol. 7, 1998, pp. 63–78. [Online]. Available: https://dl.acm.org/citation.cfm?id=1267554

[4] G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi, "SafeStack: Automatically patching stack-based buffer overflow vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 10, no. 6, pp. 368–379, Nov. 2013.

[5] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th USENIX Conf. Oper. Syst. Design Implement.*, Broomfield, CO, USA, 2014, pp. 147–163.

[6] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Proc. Workshop Program. Lang. Anal. for Secur. (PLAS)*, York, NY, USA, Oct. 2017, pp. 25–35.

[7] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *Proc. 27th Conf. USENIX Secur. Symp.*, Baltimore, MD, USA, 2018, pp. 99–116.

[8] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *Proc. 29th Conf. USENIX Secur. Symp.*, Boston, MA, USA, 2020, pp. 1111–1128.

[9] J. Vanegue, "The automated exploitation grand challenge, tales of weird machines," presented at the H2HC Conf., Sao Paulo, Brazil, Oct. 2013.

[10] Y. Younan, "Security of memory allocators for C and C++," Dept. Comput. Sci., Univ. Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW 419, 2005.

[11] P. Phantasmagoria, "The malloc maleficarum," unpublished.

[12] H. Matsukuma, "House of Einherjar–Yet another heap exploitation technique on GLIBC," presented at the CODEBLUE Conf., Tokyo, Japan, 2016.

[13] *Shellphish How2Heap-A Repository for Learning Various Heap Exploitation Techniques*. Accessed: Nov. 7, 2020. [Online]. Available: https://github.com/shellphish/how2heap

[14] Blackngel, "Malloc Des-Maleficarum," *Phrack Mag.*, vol. 13, no. 66, 2009.

[15] G. P. Zero. *The Poisoned NULL Byte*. Accessed: Aug. 2014. [Online]. Available: https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html

[16] F. Goichon, "Glibc adventures: The forgotten chunks," in *Proc. Context Inf. Secur.*, New York, NY, USA, vol. 28, Jan. 2015, pp. 1–35. [Online]. Available: https://www.contextis.com/resources/white-papers/glibc-adventures-the-forgotten-chunks

[17] D. Lea. (2000). *A Memory Allocator*. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html

[18] W. Gloger. (2006). *Ptmalloc*. [Online]. Available: http://www.malloc.de/en/

[19] C. B. Weinstock and W. A. Wulf, "An efficient algorithm for heap storage allocation," *ACM SIGPLAN Notices*, vol. 23, no. 10, pp. 141–148, Oct. 1988.

[20] *Musl Libc*. Accessed: Nov. 7, 2020. [Online]. Available: http://www.musl-libc.org/

[21] *The GNU C Library (GLIBC)*. Accessed: Nov. 7, 2020. [Online]. Available: http://www.gnu.org/software/libc/

[22] *Programming Languages-C*. ISO/IEC JTC1 SC22 WG14, ISO/IEC Standard 9899, Org. Standardization, Geneva, Switzerland, 1999.

[23] A. Sotirov, "Heap Feng Shui in javascript," presented at the Black Hat Eur., 2007. [Online]. Available: http://www.mathcs.richmond.edu/~dszajda/research/summer_2014/papers/sotirov_heap_feng_shui_javascript_paper.pdf

[24] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[25] PaX Team. (2003). *Address Space Layout Randomization (ASLR)*. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[26] M. F. Rørvik, "Investigation of x64 GLIBC heap exploitation techniques on Linux," M.S. thesis, Dept. Inform. Fac. Math. Natural Sci., Univ. Oslo, Oslo, Norway, 2019.
[27] M. Zalewski. *American Fuzzy Lop (AFL) Fuzzer-Technical Details*. Accessed: May 21, 2020. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt
[28] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Oct. 2009.
[29] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.
[30] Microsoft. (2018). *Microsoft Security Intelligence Report*. [Online]. Available: https://www.microsoft.com/en-us/security/business/security-intelligence-report
[31] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, pp. 74–84, Feb. 2014.
[32] Heelan, Sean, Tom Melham, and Daniel Kroening, "Automatic heap layout manipulation for exploitation," in *Proc. 27th Conf. USENIX Secur. Symp*, Baltimore, MD, USA, 2018, pp. 763–779.
[33] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "FreeGuard: A faster secure heap allocator," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2389–2403.
[34] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proc. 27th ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, Jun. 2006, vol. 41, no. 6, pp. 158–168.
[35] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proc. 17th ACM Conf. Comput. Commun. Secur. (CCS)*, Oct. 2010, pp. 573–584.

**SUMIN CHAE** received the B.S. degree in mathematics from the University of Seoul, Seoul, South Korea, in 2019. He is currently pursuing the M.S. degree in information security with the Graduate School of Information Security, Korea University, Seoul. His research interests include software security and memory protection.

**HONGJOO JIN** received the B.S. degree in computer engineering from Hongik University, Seoul, South Korea, in 2017, and the M.S. degree in information security from Korea University, Seoul, in 2019, where he is currently pursuing the Ph.D. degree in information security with the Graduate School of Information Security. His research interests include virtual memory protection, embedded device security, and static program analysis.

**MOON CHAN PARK** received the B.S. degree in mathematics from the University of Seoul, Seoul, South Korea, in 2013, and the M.S. degree in information security from Korea University, Seoul, in 2015, where he is currently pursuing the Ph.D. degree in information security with the Graduate School of Information Security. His research interests include applied cryptography and software security.

**DONG HOON LEE** (Member, IEEE) received the B.S. degree from the Department of Economics, Korea University, Seoul, in 1985, and the M.S. and Ph.D. degrees in computer science from The University of Oklahoma, Norman, in 1988 and 1992, respectively. Since 1993, he has been with the Faculty of Computer Science and Information Security, Korea University. He is currently a Professor and the Director of the Graduate School of Information Security, Korea University. His research interests include the design and analysis of cryptographic protocols in key agreement, encryption, signatures, embedded device security, and privacy enhancing technology.