DOI:
10.1145/3139337.3139346

Document Version
Peer reviewed version

Modular Synthesis of Heap Exploits

Dusan Repel
Information Security Group
Royal Holloway, University of London
United Kingdom

Johannes Kinder
Department of Computer Science
Royal Holloway, University of London
United Kingdom

Lorenzo Cavallaro
Information Security Group
Royal Holloway, University of London
United Kingdom

ABSTRACT

Memory errors continue to compromise the security of today's systems. Recent efforts to automatically synthesize exploits for stack-based buffer overflows promise to help assess a vulnerability's severity more quickly and alleviate the burden of manual reasoning. However, generation of heap exploits has been out of scope for such methods thus far. In this paper, we investigate the problem of automatically generating heap exploits, which, in addition to finding the vulnerability, requires intricate interaction with the heap manager. We identify the challenges involved in automatically finding the right parameters and interaction sequences for such attacks, which have traditionally required manual analysis. To tackle these challenges, we present a modular approach that is designed to minimize the assumptions made about the heap manager used by the target application. Our prototype system is able to find exploit primitives in six binary implementations of Windows and UNIX-based heap managers and applies these to successfully exploit two real-world applications.

CCS CONCEPTS

• **Security and privacy → Vulnerability management**; **Software and application security**;

KEYWORDS

Exploitation, vulnerabilities, symbolic execution

1 INTRODUCTION

Programming errors that allow the corruption of critical portions of program memory, such as stack and heap buffer overflows, remain a prevalent problem [24, 26]. An attacker can exploit such vulnerabilities and inject new code to be executed or re-use existing code for malicious purposes. Even though many modern programming languages are memory safe and rule out such risks by design, unsafe low-level languages, such as C and C++, continue to be popular. This is driven not only by large amounts of legacy code, but also performance requirements and the resource constraints of embedded environments.

Buffer overflows on the stack are well-studied and have a long history of being exploited. The basic strategy is to overflow a local buffer on the stack with input data until it overwrites a code pointer (typically the return address). An arms race of ever-more sophisticated defenses and attacks has lead to stack exploits becoming increasingly difficult to execute against hardened programs [24].

PLAS'17, October 30, 2017, Dallas, TX, USA

Tools for automated exploit generation are designed to find stack-based vulnerabilities and automatically construct customized exploits [2, 4, 6, 16]. While the appeal of such tools to potential attackers seems obvious, they actually offer a powerful pro-active defense strategy in the form of an automated penetration tester. Using these tools, developers can attempt to exploit their own systems at low cost. Furthermore, by seeding an exploit generator with a reported bug, developers can automatically assess the bug's exploitability and prioritize its patching accordingly.

Attacks against the heap are considerably more difficult than stack-based exploits. They are based on overflowing a dynamically allocated buffer and overwriting metadata, which causes subsequent operations of the heap manager (such as `free`) to violate security assumptions. For example, by writing attacker-controlled data to an attacker-controlled location. Just like stack-based buffer overflows, heap attacks require a programming error, such as a missing bounds check in the target application, which introduces the vulnerability in the first place. In addition, however, setting up the attack correctly requires intricate knowledge of the structure of heap metadata and the internal state of the heap manager; without it, the program will likely crash without executing any attacker-controlled code. Akin to the arms race in stack exploits, modern developments in hardening heap managers against common exploit techniques have made this type of attack even more complex [20], but still feasible.

So far, the task of crafting exploits for the heap–even for classic vulnerabilities in systems like Windows XP—still lies firmly in the realm of manual analysis. Despite similarities to stack-based exploit generation (e.g., the requirement of an overflow-type vulnerability), the absence of automatic techniques for heap exploits suggests that heap-specific challenges are fundamentally difficult to overcome. In this paper, we focus on the key differences between stack-based and heap-based exploit generation. Existing approaches for finding the initial overflow vulnerability can be fully reused; what differs is the search for a feasible *exploit*, given an existing vulnerability. We make the following contributions:

- We introduce heap-based vulnerabilities, in particular, the classic unsafe unlinking vulnerability, in the context of the automatic exploit generation problem. We explain the key challenges of the problem and analyze the steps required for any successful exploit in this class of attacks (§3).
- We propose a modular approach based on symbolic execution to automatically find (i) reusable attack patterns against heap managers and (ii) instances of these patterns in real-world applications (§4).
- We demonstrate our approach using a prototype implementation (§5) and present a series of experiments where we generate working exploits for binaries of both closed- and open-source heap managers and applications (§6).

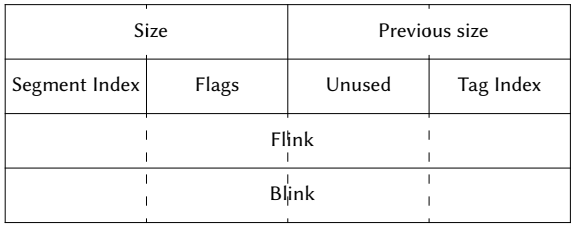

Figure 1: Memory layout of the free chunk header on the Windows heap (32 bit system). Each box corresponds to one byte.

```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) { \
    FD = P->fd;                \
    BK = P->bk;                \
    FD->bk = BK;               \
    BK->fd = FD;               \
}
```

Figure 2: The `unlink` macro from glibc 2.3.3.

2 BACKGROUND

We now first briefly recall the functionality and concepts behind heap memory management (§2.1) and then provide the basics of symbolic execution, the program analysis technique that underlies our approach (§2.2).

2.1 Heap Memory Management

The heap memory manager is the system component responsible for the provision, organization, and optimization of dynamically allocated memory. At runtime, applications request memory from the heap manager using calls such as `malloc()` or `HeapAlloc()`. The heap manager maintains a list of free memory chunks and, upon receiving a request for memory of a particular size, it searches that list for a chunk greater than or equal to that requested by the client application. It is the application's responsibility to respect the boundaries of the memory chunk and to eventually release it by calling `free()` or `HeapFree()`.

Freelists. Different heap managers select different locations for storing heap metadata. Many popular heap managers, including the default Windows heap manager [17] and Linux's `dlmalloc` or `ptmalloc2` [10], employ *freelist*-based memory management. In this model, the heap manager prefixes a memory chunk with heap metadata (a header) that describes attributes such as the flags and size of the chunk. This results in a heap layout where memory chunks containing application data are intermixed with heap metadata. If an application inadvertently permits user data to be written past the boundaries of the allocated chunk, then there is a good chance of user input overwriting adjacent heap metadata.

If a memory chunk is not allocated, then its header forms part of the freelists and contains both a forward (`fd`) and backward (`bk`) pointer to the next and previous free chunk, respectively. These headers are traversed by the heap manager while it searches for suitable chunks during memory allocation. Other operating systems, for example, FreeBSD and OpenBSD, use BiBoP memory managers [3], which align allocations to page boundaries and store metadata at the start of a page.

Windows XP Heap. In Windows XP, applications dynamically allocate memory via userspace API functions in `kernel32.dll`, such as `HeapAlloc` and `HeapFree`, which in turn forward the requests to API functions like `RtlAllocateHeap` of the Heap Manager residing in `ntdll.dll`. The heap manager is divided into a high-performance front-end manager, utilizing lookaside lists and the low fragmentation heap, and a robust, general-purpose back-end manager, utilizing freelists and the heap cache [20, 25]. The purpose of both is to minimize the amount of requests for large memory blocks that must be forwarded to the operating system's virtual memory manager (VMM).

The Heap Manager divides blocks acquired from the VMM into smaller, re-usable chunks. The heap chunk header is 16 bytes in size (see Figure 1). The first 8 bytes, containing the chunk size and flags, are present in every header type, including busy chunks, but the `fd` and `bk` pointers are only present in free chunks of memory. The back-end heap manager maintains several circular doubly-linked lists (`FreeList[0]` – `FreeList[128]`) to keep track of free memory chunks in any particular heap. When the client application calls `HeapAlloc`, the heap manager searches the freelists; if it finds a suitable chunk (equal to or greater than in size than was requested), the heap manager *unlinks* the chunk from a FreeList and returns it to the client application.

Windows versions up to XP Service Pack 1 implement the unlinking of a free chunk header `P` without any sanity checks in essentially the same way as the multi-line macro in Figure 2, which is found in the source code to `ptmalloc` in the GNU C library 2.3.3. Note that `ptmalloc` uses `fd` and `bk` in place of `flink` and `blink` for the list pointers. Arguments `BK` and `FD` are used as temporary storage.

Unsafe Unlinking Exploit Primitive. An attacker who controls `P->fd` and `P->bk` can choose their values to trigger a write of an arbitrary value to an arbitrary memory location. The line `FD->bk = BK` will write the value in `P->bk` to the address computed as the sum of `P->fd` and the offset of the `bk` field in the enclosing list struct, i.e., `(p->fd)->bk = p->bk`, in expanded form. The second write access to `BK->fd` then reverses the roles of the values; its values depend directly on the ones chosen for the first write and can trigger an access violation if not chosen carefully (this is a typical challenge for writing working heap exploits).

Such elementary write-anything-anywhere operations have been dubbed *exploit primitives*, since they serve as building blocks in a chain of primitives used to achieve arbitrary code execution. There are a number of other common heap-management operations, such as the coalescing of two adjacent free chunks into a single large chunk of memory, that may give rise to exploit primitives if heap metadata is corrupted and not verified.

Windows versions beginning with XP Service Pack 2 (SP2) have added two sanity checks to the `unlink` macro that use the data structure invariants of the circular doubly-linked freelist (`node->bk->fd == node` and `node->fd->bk == node`) to verify the list's local integrity before executing a write.

Lookaside List Exploit Primitive. Singly-linked lists, such as the lightweight lookaside lists in the Windows heap manager, do not allow to implement such a simple invariant check. Thus, versions up to Windows 2003 Server remain vulnerable via their lookaside lists even though the exploit primitive in the `unlink` operation was removed.

The lookaside list can be exploited by corrupting heap metadata such that an attacker-chosen pointer is eventually inserted into the list. Once `HeapAlloc` returns an entry from the lookaside list to the application, any write to that pointer by the application targets attacker-chosen memory. If the data written is also attacker-chosen, then the attacker has again found an exploit primitive.

2.2 Symbolic Execution

Symbolic execution is a technique for the systematic enumeration of program paths, and it has been highly successful in automated test case generation [5, 7, 13, 14]. In symbolic execution, inputs to the program under test are given symbolic instead of concrete values. Whenever a symbolic input variable is used in a conditional statement, execution forks and follows both branches. During execution, the conditional expressions on branches are added as conjunctions to the path condition. The path condition expresses the condition over input variables under which that path is taken. Whenever a path forks into two, the symbolic execution engine can rule out infeasible paths by calling a constraint solver to check whether both or just one of the resulting path conditions is satisfiable. Symbolic execution is sound, since all the paths it explores are also feasible in real executions.

In principle, a symbolic execution engine eventually explores all control flow paths in a target program; symbolic execution is theoretically complete. In practice, the exponential growth in the number of paths limits the amount of exploration that an engine can achieve. Many symbolic execution engines furthermore forego completeness by sometimes concretizing parts of the symbolic state space. For instance, when external functions are called, parameters whose value depends on symbolic input can be fixed to a single concrete value to rule out any forking in the callee.

Automatically generating exploits is in many ways similar to generating a test case exhibiting a particular bug. Therefore, symbolic execution is well-suited as a foundation for this task. Prior work on automatic exploit generation has either built directly on symbolic execution [2, 4, 6], or closely related techniques such as bounded model checking [16]. Many challenges that exploit-generation systems encounter in practice, e.g., path explosion, are also largely shared with symbolic execution. For the purpose of this paper, we treat this problem as orthogonal, but acknowledge that it is an active area of ongoing research.

Path explosion (or, equivalently, state space explosion) describes the problem arising from the fact that, in general, the number of program paths is exponential in the size of the program. Many techniques have been proposed to cope with path explosion, including search strategies that prioritize *important* paths [5], function summaries [12], and state merging, which tries to reduce the number of paths by combining states using disjunctions [18].

Interactions with the environment increase the difficulty of exercising accurate behavior in the program under test. Tools such as KLEE [5] are equipped with a handful of system call models that abstract and imitate the application-system interaction. Unlike KLEE, the S^2E system [7] does not model the environment, but instead provides a full operating system stack, composed of applications, system libraries, drivers and the kernel. If required, S^2E could explore the entire system symbolically, although in practice, one typically chooses to run most of the system concretely while just selectively enabling symbolic execution. The environment is normally several orders of magnitude larger than the unit under test and avoiding its exploration improves scalability.

3 AUTOMATIC HEAP EXPLOITATION

We now frame the problem of automatic heap exploitation (§3.1), introduce our modular approach (§3.2), and illustrate its phases following a practical example (§3.3).

3.1 Problem Definition

In the scope of this paper, we restrict ourselves to read and write exploit primitives. Therefore, we define a *heap vulnerability* as an application vulnerability that allows an attacker to manipulate heap metadata into executing an exploit primitive for writing attacker-controlled data to an attacker-controlled location. Our goal is to design an algorithm that is complete (or as complete as possible) for this subclass of heap-related vulnerabilities, and that reliably finds and exploits write primitives in heap management code.

The problem of exploiting heap-based vulnerabilities differs from that of exploiting stack-based or string-format vulnerabilities in that it actually involves two separate targets: (i) the *application* containing a heap-based buffer overflow and (ii) the *heap manager* that mediates the memory allocations. Exploit primitives in heap managers, e.g., `write-4` or `write-n` for writing 4 or n bytes to an arbitrary address, respectively, exist independently of application-specific implementations. Thus, it suffices to locate a set of exploit primitives once for each heap allocator. The exploit primitives and often even the subsequent control-flow hijack parameters can then be directly applied to different applications (given that non-randomized code, such as trampoline offsets, remains constant in shared modules such as `kernel32.dll`).

This modularity is a key aspect of our approach. For a given heap manager, we first discover *reusable exploit primitives* in an automatic process using a generic testing harness (the *application surrogate*). For a given application and runtime environment, we then use the matching primitives to automatically generate an exploit for a heap-based vulnerability.

The contributions of our present work focus only on the specifics of exploiting heap management code; we consider the general search for application bugs (which could lead to vulnerabilities) as an orthogonal problem. Indeed, existing powerful test case generation tools can provide input for our system, which then assumes the role of classifying bugs according to their exploitability. This approach is in line with previous work on exploit generation that seeds its search with known crashing inputs [2, 16].

Modern exploit mitigation techniques such as Address Space Layout Randomization (ASLR) [19] or $W \oplus X$ present further challenges that we consider out of scope for now. In addition to the path

```
77F52346      mov    [ebp-C4h], eax
77F5234C  L_unlink:
77F5234C      mov    [eax], ecx
77F5234E      mov    [ecx+04h], eax
77F52356      ...
```

Figure 3: A code segment with an exploit primitive.

```
77EB9B82      mov    eax, [L77ED63B4]
77EB9B87      cmp    eax, esi
77EB9B89      jz     L77EB9BA0
77EB9B8B      push   edi
77EB9B8C      call   eax
```

Figure 4: The UEF exception handler dispatch.

condition for reaching an exploit primitive, such defense mechanisms introduce additional constraints on the exploit solution (and may render some vulnerabilities non-exploitable). A number of techniques exist to circumvent known protections in many scenarios and may be applicable in the context of automatic exploit generation [9, 22, 23].

3.2 Modular Exploit Generation

Overall, our approach to generating heap exploits follows a five-phase process, of which the first two phases are always independent of the target application, and the third and fourth can often be kept independent. All phases rely on symbolically executing the application program together with the heap management code. The phases are:

(1) Interact: Find a crashing sequence of heap-interactions that overwrites heap metadata and generate a surrogate program that implements the sequence.
(2) Primitive: In the surrogate, fill the overflow buffer with symbolic bytes to discover an exploit primitive in the heap manager.
(3) Hijack: In either the heap manager or the target application, locate a transfer of control flow to a memory pointer and impose constraints on the symbolic input such that the exploit primitive hijacks the pointer.
(4) Bounce: If necessary for control flow diversion, locate a trampoline in library or application code that transfers control to attacker-controlled code.
(5) Payload: Synthesize the exploit payload and emit driver code for feeding it to the application.

3.3 Algorithm Walkthrough

We now present an example of applying the algorithm in order to clarify the individual steps.

Enumerate Heap Interaction Patterns. The initial step (Interact) entails finding a sequence of interactions between an application and a heap manager that permits heap metadata to be sequentially overwritten by a buffer overflow. As input, this phase uses an alphabet of heap-management functions and buffer access and overflow operators. Our system enumerates sequences until it finds one that leads to the corruption of heap metadata and a subsequent crash, e.g., (`HeapCreate`, `HeapAlloc`, `HeapAlloc`, *Overflow*, `HeapAlloc`). Here, the trailing `HeapAlloc` call trusts the now-corrupted metadata and performs an unsafe `unlink` operation (see §2.1), causing the subsequent execution of an exploit primitive. The crashing sequence is then cast into a surrogate program that acts as a test harness for the heap manager.

The output of the Interact phase is a set of similar interaction sequences that lead to exploit primitives and vulnerable heap layout configurations for arbitrary heap managers.

Find Exploit Primitives. During the second phase (Primitive), our system makes the overflowing bytes symbolic, because they will be derived from input in a concrete attack. It then monitors the program state for exploit primitives, which it detects as writes of symbolic data to a symbolic address. This is synonymous with a write of attacker data to an attacker-controlled location.

In our example, a memory copy instruction with symbolic operands is observed in `ntdll.dll` (see Figure 3). Both the `EAX` and `ECX` registers are symbolic and can be freely chosen by the attacker (modulo the constraints imposed by the path condition).

At this point, our system has produced a path condition that corresponds to a range of concrete inputs under which the target program reaches an exploit primitive.

Control Flow Hijack. To achieve arbitrary code execution, the exploit must divert the control flow of the application. In phase Hijack, our system uses the exploit primitive to overwrite the memory location that will be used for the next indirect control transfer reachable from the exploit primitive.

Examples of exploitable indirect control transfers are function pointers or installed exception handlers. In our example, the corruption of heap metadata causes a second exploit primitive to produce an access violation in the heap manager and triggers a series of exception handlers. As Figure 4 shows, one of the exception handling dispatch routines moves the value at memory address `77ED63B4` into `EAX` and calls it.

This memory address serves as the target for the exploit primitive and completes the next step in the chain; by constraining `EAX` to be equal to `77ED63B4` when executing the `mov [eax], ecx` instruction in Figure 3, control will eventually transfer to the location pointed to by `ECX`.

Trampoline Search. To complete the exploit, we need to constrain the manipulated jump target such that the program executes foreign code, which, in our exploit model, is held in the attacker-controlled overflow buffer. To make this control transfer reliable, Bounce searches for a "trampoline", another indirect control transfer, that jumps to the address held in a register that happens to point to the user-supplied buffer at the time.

Continuing our example, our system scans the processor state at the time of the initial control flow hijack and finds that the registers `EDI` and `EBP` point to our buffer. The system then searches for a matching trampoline, i.e., `call` or `jmp` instructions to `EDI`/`EBP`+*offset*, in any module loaded in the target process. This address completes the next step; the `ECX` register has to be constrained to the trampoline memory address when the exploit primitive is executed.

Exploit Generation. The remaining phase, PAYLOAD, consists of constructing a valid shellcode that satisfies the constraints imposed on the buffer at the time of the trampoline jump. We use an elastic shellcode template with `NOP` slides that can be adapted to satisfy constraints. It is fitted with Service Pack-specific offsets to API functions called from the shellcode. The exploit is expressed as a C-based character array and also packaged into a stand-alone executable Python script, selected according to the attack vector's method of delivery, e.g., over a network or the command line.

4 THE HEAP EXPLOIT SYNTHESIS CHAIN

We now present the details of our approach and explain the phases for finding and satisfying the constraints under which the full chain of events, necessary for the successful construction of an exploit, unfolds. We discuss how to discover a vulnerable heap interaction sequence (§4.1), locate a suitable exploit primitive (§4.2), hijack the application control flow (§4.3), find a suitable trampoline (§4.4), and synthesize the final exploit (§4.5).

4.1 Application-Heap Interaction

Given the implementation of an arbitrary heap manager H and its API, the first phase (INTERACT) consists of searching for a sequence of application-heap interactions that corrupts heap metadata and violates the internal consistency of heap data structures. Once a crashing sequence is found, it is written to a surrogate program, which is then passed to the next phase, PRIMITIVE.

Successful sequences will typically contain (1) a call for creating a private heap (if necessary), (2) initial memory allocations to generate metadata and a target memory buffer on the heap, (3) an overflow θ to overwrite metadata, (4) an in-bounds write γ (if necessary) and (5) a heap API call processing the invalid metadata and triggering an exploit primitive. However, the exact sequence and number of events that is required depends on the particular heap manager. With this knowledge, it is possible to guide search heuristics by prioritizing promising sequences of heap interactions.

This set of operations has so far been sufficient, but could in principle be extended by other generic operations. While the API calls and the overflow operator have obvious purposes, the need for an in-bounds access is more subtle: depending on the type of heap metadata corruption, an exploit primitive may only be reached once the application executes a normal write (of attacker-controlled or constant data) to a corrupted pointer, which has in turn been inadvertently returned by a heap API function.

INTERACT begins by running an application surrogate that is an interpreter for the alphabet

$$\Sigma = \{\texttt{Create}, \texttt{Alloc}, \texttt{Free}, \gamma, \theta\},$$

where `Alloc` and `Free` stand for `malloc` / `HeapAlloc` and `free` / `HeapFree`, respectively, `Create` for the Windows-specific call to `HeapCreate`, γ for an in-bounds buffer write, and θ for an out-of-bounds heap overflow. The interpreter iterates over an input string, interpreting symbols until it reaches the end and exits. For each heap-management symbol, it executes the corresponding operation.

The input string can be fuzzed or symbolically executed up to a fixed length (our prototype does the latter). With five symbols in Σ, injecting a string of three characters results in a search space of $5^3 = 125$ heap interactions. Crashes can be robustly detected by interpreting signals (on Linux) or intercepting the `UnhandledExceptionFilter` (on Windows).

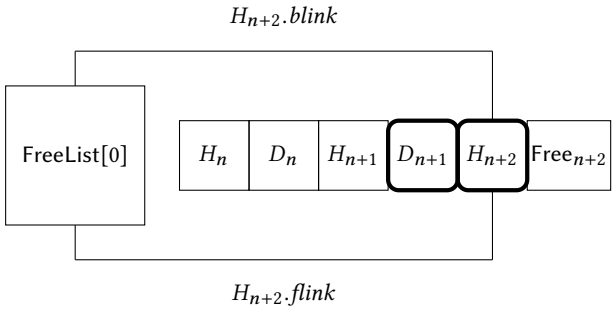

Figure 5: Heap metadata is adjacent to buffer content.

Heap Configurations. In Windows heap management, after allocation requests for memory of size D_n and D_{n+1} bytes, and with $Free_{n+2}$ bytes remaining unallocated in the heap, the memory layout will resemble that of Figure 5. Header H_{n+2} references a free block of memory and forms part of the FreeLists. If an application permits buffer D_{n+1} to be overflown (the overflow area is marked in bold), then the `flink` and `blink` pointers in H_{n+2} can be set to arbitrary values. Header H_{n+2} points back to the `FreeList[0]` such that a search for available memory terminates upon returning to the beginning of the head node. In Figure 5, the heap is not fragmented and coalescing is not required, so H_{n+2} can summarize the entirety of free memory available in the heap. Any further allocations would split $Free_{n+2}$ into D_{n+2} and $Free_{n+3}$, moving H_{n+2}'s `flink` and `blink` pointers further towards the end of the heap. However, a series of de-allocations could poke holes in consecutively allocated memory and would result in a fragmented heap, with buffer D_n potentially sitting next to new `flink` and `blink` pointers. The ability to conduct more advanced manipulations of heap memory layouts is desirable, because it can enable more surgical heap exploitation.

Limitations. In this work, we restrict our model to heap-based buffer overflows that always overwrite heap metadata sequentially by writing past the boundaries of allocated buffers. However, in practice, there exist many methods for overwriting heap metadata. For example, an integer arithmetic error in an array subscript could directly corrupt heap metadata from any point in a program, while leaving adjacent fields, such as heap header cookies, intact.

4.2 Heap Exploit Primitives

Given the heap manager H and a surrogate S implementing a known-crashing interaction sequence, the next phase, PRIMITIVE, discovers a set of heap exploit primitives P for overwriting security-sensitive data in the application.

Figure 6 shows the set of exploit primitives with respect to symbolic bytes. $\mathcal{M}_n[\cdot]$ maps a memory address to its corresponding n-byte value and x is an attacker-controlled symbolic value, which may have arbitrary constraints imposed upon it. If only attacker-specified input is made symbolic and critical operations eventually manipulate symbolic bytes, then attacker input is reaching critical

$\mathcal{M}_n[c] \leftarrow x$	symbolic write-n to fixed location
$\mathcal{M}_n[x] \leftarrow c$	fixed write-n to symbolic location
$\mathcal{M}_n[x] \leftarrow x$	symbolic write-n to symbolic location
$v \leftarrow \mathcal{M}_n[x]$	read-n from symbolic location

Figure 6: Description of heap exploit primitives.

```
if(!prev_inuse(p)) {
    prevsize = p->prev_size;
    size += prevsize;
    p = chunk_at_offset(p, -prevsize);
    unlink(p, bck, fwd);
}
```

Figure 7: Coalescing of chunks in dlmalloc.

operations under some constraints. The constraints determine the level of control that the attacker exercises over the values used in those critical operations. Hence, once a flow of symbolic data to a symbolic destination is detected, we have discovered a heap exploit primitive. Generally, we deal with *full* or *partial* `write-n` primitives (for $n = 1$, 2, or 4 or a 32-bit system). Full `write-n` primitives give the attacker control over both the data and the destination address, whereas partial writes only allow the attacker to control one of them.

Primitive involves injecting symbolic data past the boundaries of allocated buffers, as per the application-heap interaction sequence determined in Interact, and repetitively picking new paths to explore in surrogate S (see Algorithm 1). Each path is executed, instruction by instruction, until program termination or until an exploit primitive is found. If the instruction is of the form $\mathcal{I} = (\mathcal{M}_n[A] \leftarrow V)$, such that a value V is being written to memory address A, and both A and V contain symbolic values, then the instruction $\mathcal{I}$ is a write-n primitive.

The path is passed to the the next phase, Hijack, and execution resumes from the instruction immediately following the exploit primitive.

Read Primitives. Some heap managers, such as `dlmalloc` and `ptmalloc2`, also require the use of *read* exploit primitives. Upon overflowing the heap chunk header with symbolic bytes, field `p->prev_size` becomes symbolic (see Figure 7) and the `unlink` macro performs memory load operations from the symbolic expression. Depending on the memory model of the symbolic execution engine used, a symbolic read is either concretized or leads to expensive subsequent solver queries involving array logic. We use a concrete memory model, i.e., a symbolic expression must be concretized before it is used as a pointer for a memory read. Conceptually, any feasible address is a possible solution; for completeness, all possible addresses have to be eventually enumerated. We decide to concretize symbolic reads to a memory address within bounds of the attacker-controlled buffer, if possible. This follows a general strategy of making symbolic as much as possible of the program state. If the value chosen does not lead to a write primitive, the current path terminates unsuccessfully and a new path is forked with a new value. In the case of `dlmalloc`, the result is that the `unlink` macro fetches symbolic bytes and ultimately executes a write-4 exploit primitive as before.

Data: a surrogate S exercising a select sequence
Result: a tuple $\{A_{val}, V_{val}\}$ for exploit primitive

```
while (P = pickNewPath(S)) ≠ ⊥ do
    while (I = nextInstruct (P)) ≠ ⊥ do
        if (I = M_n[A] ← V) then
            if (A = sym) ∧ (V = sym) then
                {A_val, R_ef} = Hijack (P, I);
                V_val = Bounce (R_ef);
                ok = P.aC(A = A_val, V = V_val);
                if ok ≠ ⊥ then
                    return {A_val, V_val};
                end
            end
        end
    end
end
```

Algorithm 1: Discovering an exploit primitive.

4.3 Hijacking the Control Flow

Hijack addresses the problem of finding, given a set of exploit primitives, a writable pointer T such that a single or a chain of exploit primitives can hijack the control flow of the heap manager by redirecting T to an attacker-controlled address.

To this end, Hijack locates indirect control transfers on the current path that depend on writable memory locations using a static dependency analysis. For simplicity, we focus on locations that are not modified (note that aliasing does not pose a problem in path-wise symbolic execution with a concrete memory model). If the path does not contain an indirect control transfer, the current path will again terminate unsuccessfully and symbolic execution will resume with another path through the program, until an exploit has been either found or the program is shown to be immune to exploitation under the model used.

Hijack returns the tuple $\{A_{val}, R_{ef}\}$ where A_{val} is the memory location that control has been transferred to and R_{ef} is a relative address, such as `dword ptr[edi+74h]`, that references the injected buffer at the point of control transfer to A_{val}. Such control transfers are often observed in application-specific code, such as call tables and C++ vtables, and also exception handling routines. It is common practice in manual exploitation to build more reliable exploits by making use of `jmp` or `call` trampolines [16], rather than guessing or hardcoding memory addresses. The assumption is that a register, which happens to contain a pointer to an attacker-controlled buffer at the time of control being transferred to an arbitrary attacker-chosen address, will always contain such a pointer, regardless of the absolute value of the buffer's memory address.

4.4 Locating a Trampoline

Subsequently, R_{ef} is converted into a binary sequence that performs a `call` or a `jmp` to R_{ef} and Bounce searches for the binary sequence in all modules that are loaded in the target process (including system libraries such as `kernel32.dll` on Windows). The resulting offsets

V_{val} and A_{val} are candidate values for the write-n primitive. In other words, setting ecx to A_{val} and eax to V_{val} in a primitive such as mov [ecx], eax is, under the chosen path, guaranteed to transfer control to V_{val}, which in turn, and by construction, transfers control to an attacker-supplied buffer on the heap. To verify the suitability of the chosen values for V_{val} and A_{val}, the constraints $A = A_{val}$ and $V = V_{val}$ are added to the path condition, and, if it remains satisfiable, the values are valid for use.

At the point of the control transfer to A_{val}, all eight general purpose registers are scanned for values falling within the range of the injected buffer. For each register r, our system also performs a scan from dword ptr[r+00h] to dword ptr[r+FFh] to locate indirect references to the buffer.

Finally, a working exploit requires finding memory offsets for API functions used in the shellcode and call trampolines that redirect control to shellcode. To that end, Bounce uses an in-vitro scanner embedded in the guest operating system to scan modules of interest. The list of modules is compiled from the modules that are loaded in the target process at the point of the control flow hijack.

4.5 Synthesis of Exploit Payload

The final phase, Payload, generates a shellcode respecting all constraints established in the previous phases. Given an application containing a heap-based buffer overflow and a set of exploit-friendly heap-interaction sequences, the generated exploit has to guide the application towards one such sequence. In addition, the exploit payload must cause an exploit primitive to overwrite an invoked pointer and cause subsequent execution of arbitrary code.

Recall that a trampoline transfers control to a memory address residing within the boundaries of the injected buffer. Hence, the exact offset from the start of the buffer, to which control is transferred, is dependent on R_{ef}. We shall refer to the bytes residing at that offset as the *landing site* (see Figure 8). An exploit must be constructed within the confines of both *spatial* and *value* limitations on the input. Since R_{ef} transfers control to a particular offset from the start of the injected buffer, it is mandatory to exercise control over several bytes at the landing site. If the successive bytes are *bad bytes*, this at least permits us to introduce a jmp instruction to the rest of the shellcode. Failure to do so could cause an invalid instruction or access violation once control reaches that part of the buffer. In order to avoid executing bad bytes in the user input that cannot, due to constraints, assume values of valid instructions, we prefix all such bytes with a jmp and conveniently jump over them. If we install shellcode as an exception handler, an invalid instruction in the shellcode may result in an infinite loop.

The rest of the bytes that do not form part of the shellcode or any auxiliary gadgets are set to NOP instructions in order to form a NOP slide directed towards the shellcode. The resulting NOP slide could be contiguous up to the shellcode or alternatively, it could be a segmented NOP slide. The reliability of the R_{ef} offset thus determines the probability of successfully executing the shellcode.

We use a roughly 20-byte shellcode to run calc.exe using WinExec and terminate the target process using ExitProcess. The offsets of these functions, which are Service Pack-specific, are retrieved during Bounce and are inserted into the shellcode. We require two bytes for every jmp instruction that is inserted (see Figure 8). As a consequence, a single byte located between two bad bytes is itself considered a bad byte, as it cannot facilitate a jump to valid code. Thus, the exploit is constructed using an elastic shellcode template.

```
{0x90,0x90,0x90,0x90,0x90,0x90,
 0x90,0x90, landing ,0x90,0x90,
 0x90,0x90,jump 2,bad,bad,0x90,
 0x90,0x90,0x90,0x90,0x90,0x90
 0x90,0x90,0x90,   shellcode };
```

Figure 8: An elastic exploit template.

5 IMPLEMENTATION

We designed our system as plugins (about 5 KLOC of C++) to the S^{2}E binary symbolic execution framework [7]. S^{2}E executes code manipulating only concrete values natively and dynamically translates symbolic code from x86 to LLVM bitcode for symbolic execution with KLEE.

Our S^{2}E analyzer plugin inspects program states for heap exploit primitives. We also make use of a custom selector plugin to apply search heuristics, such as path prioritization. In addition, we have extended S^{2}E plugins, such as the WindowsMonitor plugin, to work on unsupported Windows XP service packs, e.g., SP0 and SP1. The purpose of the extensions is merely to allow S^{2}E to run Windows XP SP0 and SP1 as guest operating systems and is not related to the technique presented in this paper. We also modified KLEE's core modules to enable the partial processing of concolic bytes and floating point data types. The modification proved to be crucial in attacking the GDI component in Windows (see §6.2). In the exploit generation phase, we produce a compact stand-alone Python script that delivers the exploit over a chosen interface, e.g., over TCP/IP sockets to network-enabled applications.

6 EVALUATION

In this section, we first present our evaluation targets and methodology (§6.1) and then present experimental results to answer the following questions:

(1) Effectiveness (§6.2): Can our system automatically generate heap exploits for real-world applications?
(2) Generality (§6.3): Does our system apply to a wide range of heap managers?
(3) Automation (§6.4): What level of automation does our implementation offer?
(4) Performance (§6.5): What is our system's overall performance and what is the contribution of the individual steps?

6.1 Evaluation Targets and Methodology

Heap Managers. As target heap managers, we selected all four Windows XP heap managers, from Service Packs 0 to SP3, and the open source implementations of dlmalloc (Doug Lea's malloc) and ptmalloc2 (the heap manager currently used in the GNU C library, glibc). The evolution of the security of the built-in Windows

XP heap manager over the range of Service Packs is representative of the development of countermeasures across other platforms as well. The heap vulnerabilities are not mere programming errors, but complex operations on data structures which occasionally result in unsafe program states. For example, both the Windows heap and glibc contained unsafe unlink macros (see §2.1). Over the years, both gradually introduced similar safety measures, e.g., cookies to the heap header and non-writable guard pages to prevent cross-page overflows. For the purposes of exploit generation, each Windows XP Service Pack represents a completely separate heap manager, since each is a binary build with a unique set of pointer offsets. Consequently, an exploit is tailored for deployment against a particular Service Pack.

We also built `dlmalloc` and `ptmalloc2` on Windows, but the detection and use of their respective exploit primitives happens completely inside the code of the application. While their hijack on Windows is mediated via the UEF exception handler, a different (possibly application-specific) function pointer can serve as a hijack target on other platforms.

Applications. As test targets we employ two real-world closed-source applications, WellinTech KingView and a Windows GDI component. Both applications contain remotely exploitable heap-based buffer overflow vulnerabilities that may lead to arbitrary code execution. Manual exploits for both applications are available in online security databases.

WellinTech KingView 6.53 (CVE-2011-0406) is a SCADA/HMI application used in industrial control systems to visualize process. It is a large and complex applications consisting of hundreds of files and utilities. The vulnerability, which was discovered in 2011 and given CVE-2011-0406, is present in the `HistorySvr.exe` module that starts up in the background as a Windows service and listens on TCP port 777.

The MS04-032 vulnerability is present in a core component of the Windows operating system, the Graphics Device Interface (GDI) library. The vulnerability is triggered when the thumbnail icon of a specially-crafted Enhanced Metafile (`.emf`) image file is rendered by an application. An attack vector would include an HTML email, an ordinary website or a remote shared drive.

Both real-world applications were tested on Windows XP SP1 and targeted via the `unlink` exploit primitive. The exploit generation should therefore work successfully on any of the unsafe `unlink` heap managers.

6.2 Effectiveness

We have successfully found and utilized fully-controlled `write-4` primitives on Windows XP SP0 and SP1; a combination of `read-4` and `write-4` primitives that work in concert with each other in `dlmalloc` and `ptmalloc2`; and partial `read-4` and `write-4`s, followed by an alphabet-induced `write-4` (full or partial) in Windows XP SP2 and SP3. The fact that a `HeapAlloc` call returns a symbolic pointer during the `lookaside` sequence means that even API hooks can recognize this vulnerability. In our model, we recognize the vulnerability, since it results in a write primitive, due to a trailing γ (within-bounds write) at the end of the sequence. In summary, we have verified applicability of our unlink attack sequence on UNIX-based systems for `dlmalloc 2.7.2` and `glibc v2.3.3` (`ptmalloc2`); on Win32 systems for Windows 2000, Windows XP SP0, and Windows XP SP1. We verified the lookaside attack on Windows XP SP2 and SP3, and Windows 2003 Server.

Our prototype system successfully automates the entire end-to-end process of crafting a `calc`-spawning exploit for the two target applications. It demonstrates that, at least for these case scenarios, the "hacker mind" can be imitated to a practical degree. For a bare-bones surrogate application, full exploit generation for an unlink vulnerability with a UEF handler hijack took 5.9 seconds; a lookaside list exploit with app-specific hijack took 9.8 seconds.

6.3 Generality

As mentioned in §6.2, we can find and utilize fully- or partially-controlled `read` and `write` primitives on all Windows XP Service Packs. In `dlmalloc` and `ptmalloc2`, successfully dealing with `read` is a pre-requisite for employing `write` primitives to hijack pointers.

Hijack Method. Our search for an invoked, writable code pointer on Windows XP SP0 and SP1 results in finding and hijacking the `UnhandledExceptionFilter`. The `dlmalloc` and `ptmalloc2` managers are compromised via the same mechanism, as neither employs its own exception handling and each passes control directly to the `UEF` after an access violation. We are, however, unable to exploit applications that preclude the execution of `UEF`, for example, by installing a `VEH` handler. The `VEH` exception handler is not the default handler and its dispatch is protected from execution by a conditional guard. This means the head node to its exception handler chain cannot be found using our method.

The hijack method slightly differs for later Windows versions. From Windows XP SP2 onward, the `UEF` pointer is protected by `EncodePointer`, rendering the `UEF` hijack method infeasible. However, unlike the `unlink` technique, the `lookaside` technique allows control flow to exit the heap manager, permitting us to search for a hijackable pointer inside application code. Thus, to hijack applications on Windows XP SP2 and SP3, we apply the same routine that detects the `UEF` dispatch to application code, automatically lifting a valid, but non-reusable target pointer.

Memory Wrappers. Often enough, mid-sized or large software projects, like the cross-platform Webkit, opt to employ their own memory-management routines, usually in an effort to achieve greater performance. We use `dlmalloc` and `ptmalloc2` as memory wrappers around the Windows heap. This scenario serves to show off that our system can exploit custom heap implementations, even if the underlying operating system heap is immune to attack. While `dlmalloc` and `ptmalloc2` are open source, our system does not use their source code as an input. We are therefore able to demonstrate that the binaries of `dlmalloc` and `ptmalloc2` on Windows can be executed symbolically, which is a pre-requisite for automatic exploit generation.

Applicability. Although our evaluation is performed on Windows XP, the exploitation techniques found and exercised by our system are also known to be applicable to Windows 2000 SP0–SP4 and Windows 2003 Server. This includes, at minimum, another five real-world heap managers that our system can target without modification. The early Windows XP versions, `dlmalloc`, and `ptmalloc2` are all attacked using the `unlink` method, as it is convenient and

sufficiently powerful. Nevertheless, our techniques are not limited to the `unlink` method, as shown by using the `lookaside` method against later Windows XP versions that are explicitly hardened against unsafe unlinking.

The benefits of our prototype system are most clear-cut when an exploit, which is under construction for a newly-tasked heap manager, differs only in minor low-level detail and is still covered by the model in use. The extending of exploit models or templates requires human reasoning, but minor low-level details are parsed in a straightforward fashion by laborious, repetitive calculations, perfectly suited for out-sourcing to a fast, automated process.

Sequence Enumeration. Designing or evolving effective heuristics to filter out non-exploitable sequences has been left for future work. The ascertaining of correct values for performing more complex heap manipulations, such as repairing the default process heap automatically, is also beyond scope. However, in all our test cases, the path from the post-overflow invocation of the `HeapAlloc` or `malloc` call to the execution of the exploit primitive was quite short. Thus, while it may not qualify as a general criterion, terminating the exploration of a sequence after 15 seconds is an effective search heuristic for isolating the `unlink` and `lookaside` sequences.

We have conducted searches of state spaces of up to 5^7 configurations, covering just over 65,000 states, which encompass both the `unlink` and `lookaside` exploitation techniques. Note that for maximum speed, one should instead employ a userland fuzzer with additional optimization steps that reduce the size of the state space. Our search lazily explores most permutations of the alphabet, including sequences without any θ operator. Using an S^2E plugin for searching, one complete sequence exploration takes on average 1.1 seconds, with θ interpreted as a concrete overflow.

Length	States	Crashes	Time (s)	Technique
1	5	0	0	
2	25	1	18	
3	120	6	94	
4	580	28	580	unlink
5	2,792	124	3,062	
6	13,468	548	11,106	
7	65,152	2,446	73,606	lookaside

Table 1: Number of states, crashes and time taken for each step in enumerating vulnerable heap interaction sequences. The unsafe unlinking attack is discovered after four steps, and the lookaside vulnerability after seven.

6.4 Automation

Injection Models. As briefly mentioned in §5, in order to simulate user input, we inject symbolic data by utilizing conventional input vectors, such as arguments, files on disk, network transmissions or environment variables. To this end, we implement a number of complex interfaces, which we have observed to be necessary for the injection of real-world applications. These complex interfaces ensure a target application receives the symbolic input properly. Our plugin intercepts `WSAAsyncSelect` in order to retrieve the message code and socket identifier used for the registration of asynchronous network event notifications. The collected data is replayed into an application's main message loop using `GetMessageA`; this simulates a network event occurrence that results either in the acceptance of a new connection or in the reading from an established connection stream. In the latter case, a `ioctlsocket` call is intercepted to simulate data waiting to be read from the operating system's network buffer. Only then is any subsequent attempt to read the data using `recv` utilized to inject symbolic bytes.

This procedure was used to inject the WellinTech KingView SCADA/HMI application. It is infeasible to deliver an oversized input to KingView, and thus infeasible to exploit it, if only `recv` is modeled. This demonstrates how difficult it is, in practice, to stimulate behavior from real-world applications. It requires not only having models for each of the four individual API calls, but also to have the four API calls work in concert with each other to create a consistent illusion of incoming network traffic.

To exploit the two real-world applications, we needed to bootstrap the symbolic execution engine with a concrete prefix and suffix. We consider finding the path to a vulnerability to be an orthogonal problem, but acknowledge that it is an active research area and an important sub-problem in a full exploit generation system.

KingView Vulnerability. To tackle the CVE-2011-0406 vulnerability in KingView, we provided an auxiliary concrete input consisting of 30,000 concrete bytes, with the addition of 70 symbolic bytes. The auxiliary bytes that form the prefix are derived from a crashing test case (without exploit). The prefix allows to reach the location of the crash without re-exploring the entire application.

The `nettransdll.dll` that is host to the heap-based buffer overflow unfortunately computes a cyclic redundancy check (CRC16) on received network data before passing it on. The error-checking calculation has no effect on the exploitability of the vulnerability, i.e., the resulting checksum does not have to match the expected value for the exploit to work. However, the execution of the CRC16 routine itself can be problematic. A concrete prefix is often employed to get the symbolic execution engine through problematic portions of code, e.g., an application is made to perform difficult computations on a concrete header of a packet, so it thereafter passes the entire packet, which bears a trailing symbolic suffix, to the code of interest. In CVE-2011-0406, a checksum is computed on the entire packet, resulting in a fork explosion upon the injection of only a single symbolic byte. Cryptographic code, e.g., message digest functions, is well-known to be problematic for symbolic execution tools. Therefore, we solve the problem by providing an S^2E abstraction for the CRC16 function with local consistency. Alternatively, a concolic string seeded with the concrete prefix can be used instead. Overall, generation of a full exploit took 22 seconds.

Windows GDI Vulnerability. To generate an exploit for the MS04-032 Windows GDI vulnerability, we provided an Enhanced Metafile (EMF) file format template as the auxiliary concrete input. The template consists of a 64-byte concrete prefix, the file header, and 4-byte concrete suffix, the file terminator. An arbitrary number of symbolic bytes (in our case, 67 symbolic bytes) was injected into the "data" portion of the EMF template by `ReadFile` hooks that intercepted the `IStream::Read` interface data buffering. The control

Technique	States	CpuConcr	CpuKlee	Queries	QConsts	UserTime (s)	QueryTime (ms)	SolverTime (ms)
Unlink (SP0)	1	190,898	0	0	0	6.87	0	0
	3	24,099,316	84	2	19	1.23	0.011	0.005
	7	24,097,122	2,315	262	2,772	1.36	0.301	0.008
	7	24,097,122	2,315	264	3,817	1.39	0.306	0.012
Lookaside (SP2)	1	231,020	0	0	0	7.48	0	0
	5	50,048,788	2,073	8	86	1.80	0.017	0.018
	6	50,779,813	5,266	12	146	1.90	0.020	0.029
	6	54,470,030	8,892	26	1,273	2.26	0.056	0.035
	6	55,675,071	8,892	27	1,322	2.43	0.059	0.038

Table 2: Number of states, executed concrete and symbolic instructions, solver queries, constructs, running time, time for query generation, and overall solver time.

flow subsequently descended into `gdiplus.dll`, whereby KLEE attempted to invoke the external function `int32_to_floatx80` with symbolic arguments. Recall that S^2E converts translation blocks that manipulate symbolic bytes into LLVM, for execution by KLEE. Vanilla KLEE does not support the invocation of the external function with symbolic arguments and only had limited experimental support for concolic data types. Thus, a few of KLEE's Core modules were patched to enable S^2E to ingest x86 floating point operations with concolic floating point data types. This enabled the end-to-end construction of exploit code for MS04-032. There is reason to suspect that future exploit systems for graphics-processing code with an S^2E back-end will demand analogous extensions. Exploit generation took 20 seconds in this case.

6.5 Performance

All experiments were performed on a 2.5 GHz Intel Core i5 with 8 GB 1600 MHz DDR3, running a Mac OS X 10.8.5 operating system. Table 1 shows statistics of our experiment in finding vulnerable heap interaction sequences (Interact). The unlink and lookaside techniques were found automatically at length 4 and 7 of the interaction string (see §4.1).

In Table 2, we show statistics over time for executing the `unlink` technique on Windows XP SP0 and the `lookaside` technique on Windows XP SP2. The number of instructions (both concrete and symbolic) give a measurement of the size of the heap manager; the number of queries estimates the effort required for symbolic execution to pinpoint the exploit primitive. We also list timing measurements for the time spent constructing queries and solving them (using the STP solver). If a system is faced with particularly complex constraints then this will reflect in the increase in time that is spent generating and solving SAT queries. None of the heap managers we tested gave rise to complex symbolic expressions, since in neither case did the symbolic bytes go through any conversion process, e.g. a hash function. This is understandable, as being critical components of operating systems, heap managers strive for best performance and simplicity. Therefore, the SAT queries produced by shellcode-building code were straightforward to solve.

7 RELATED WORK AND DISCUSSION

Automatic exploit generation tools described in academic literature [2, 6, 16] have previously tackled the problem of automating the exploit writing pipeline for stack-based buffer overflow and format string vulnerabilities. Due to limitations in their modeling of security vulnerabilities, the capability of the aforementioned systems did not extend to other classes of vulnerabilities. There is no previous study in academic literature that tackles the problem of synthesizing exploits for heap vulnerabilities. In [15], an input is produced that causes a heap-vulnerable program to crash. The result is analogous to that achieved by a fuzzer and requires no modeling or comprehension of the heap domain, nor does it require the selection of appropriate pointers to craft working shellcode.

While we have not previously observed any such instances, it is conceivable to imagine a hardened heap implementation that would pro-actively attempt to resist symbolic execution [11, 21]. Such a defense might not hinder manual efforts to construct exploits for heap implementations, but might present a challenge to automated analysis and exploit-generating tools.

Our compositional approach to heap exploitation is reminiscent of algorithms for compositional symbolic execution [1, 12]. Standard symbolic execution re-explores a procedure if two distinct paths lead through it. In contrast, compositional symbolic execution explores procedures in isolation and combines inter-procedural paths to form a set of realistic program paths. Since each intra-procedural path is explored only once, the number of possible inter-procedural paths grows linearly rather than exponentially in the number of procedures explored [12].

The most common method for tackling the state space explosion problem is restricting the size of the state space to be searched (at the risk of further incompleteness). In the implementation of existing automatic exploit generation systems [2, 6], *pre-conditioned* symbolic execution is used to narrow down the target state space to search in accordance with a chosen pre-condition. Similarly, we use concrete prefixes in demonstrating exploit generation for our real world targets.

Automated software testing has a variety of potential applications, which can be broadly characterized as either *informative*, *defensive* or *offensive*. Informative testing discloses a bug or security vulnerability within the program under test, most popular with

tools aimed at developers, such as as static analyzers or fuzzers. It is often not necessary to produce a shell-spawning exploit in order to recognize that a vulnerability is present and demands fixing. In contrast, automated defensive and offensive solutions take action in response to the discovery of a vulnerability. For example, an automated patch generator [8] aims to shorten the vulnerability window that exists from the discovery of a vulnerability to the formulation of a patch-based fix. While some degree of automation has been achieved in academic literature, end-to-end self-healing software is the subject of ongoing research.

8 CONCLUSIONS

The problem of automatically synthesizing exploits for heap vulnerabilities has not been previously tackled. In this paper, we have introduced the nature of heap-based vulnerabilities in the context of the automatic exploit generation problem. We have presented a general framework for discovering exploit primitives in heap managers with varying heap layouts. Finally, we have demonstrated that it is feasible to use our solution for real-world implementations of heap managers, and to generate working exploits for target applications.

ACKNOWLEDGMENTS

This work was in part supported by EPSRC grant EP/L022710/1. Dusan Repel was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security (EP/K035584/1).

REFERENCES

[1] Anand, S., Godefroid, P., and Tillmann, N. Demand-driven compositional symbolic execution. In *TACAS* (2008), pp. 367–381.
[2] Avgerinos, T., Cha, S. K., Hao, B. L. T., and Brumley, D. AEG: Automatic exploit generation. In *NDSS* (2011).
[3] Berger, E. D. Heapshield: Library-based heap overflow protection for free. *UMass CS TR* (2006), 06–28.
[4] Brumley, D., Poosankam, P., Song, D. X., and Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy* (2008), pp. 143–157.
[5] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), pp. 209–224.
[6] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy* (2012), pp. 380–394.
[7] Chipounov, V., Kuznetsov, V., and Candea, G. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS* (2011), pp. 265–278.
[8] DARPA. Cyber grand challenge. *Queue 10*, 1 (2012), 20.
[9] Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA* (2009), pp. 88–106.
[10] Ferguson, J. Understanding the heap by breaking it. In *Black Hat USA* (2007).
[11] Futoransky, A., Kargieman, E., Sarraute, C., and Waissbein, A. Foundations and applications for secure triggers. *ACM Transactions on Information and System Security (TISSEC) 9*, 1 (2006), 94–112.
[12] Godefroid, P. Compositional dynamic test generation. In *POPL* (2007), pp. 47–54.
[13] Godefroid, P., Klarlund, N., and Sen, K. DART: directed automated random testing. In *PLDI* (2005), V. Sarkar and M. W. Hall, Eds., ACM, pp. 213–223.
[14] Godefroid, P., Levin, M. Y., and Molnar, D. A. Automated whitebox fuzz testing. In *NDSS* (2008), The Internet Society.
[15] Hao, B. L. T. Automatic heap exploit generation. Bachelor's thesis, Carnegie Mellon University, 2012.
[16] Heelan, S. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. Tech. rep., University of Oxford, 2009.
[17] Kath, R. Managing heap memory. http://msdn.microsoft.com/en-us/library/ms810603.aspx, Apr. 1993.
[18] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient state merging in symbolic execution. In *PLDI* (2012), ACM, pp. 193–204.
[19] Li, L., Just, J. E., and Sekar, R. Address-space randomization for Windows systems. In *ACSAC* (2006), pp. 329–338.
[20] McDonald, J., and Valasek, C. Practical Windows XP/2003 heap exploitation. In *Black Hat USA* (2009).
[21] Sharif, M. I., Lanzi, A., Giffin, J. T., and Lee, W. Impeding malware analysis using conditional code obfuscation. In *NDSS* (2008).
[22] Sotirov, A. Heap feng shui in JavaScript. *Black Hat Europe* (2007).
[23] Sotirov, A., and Dowd, M. Bypassing browser memory protections in Windows Vista. In *Blackhat USA* (2008).
[24] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy* (2013), IEEE Computer Society, pp. 48–62.
[25] Valasek, C. Understanding the low fragmentation heap. In *Black Hat USA* (2010).
[26] van der Veen, V., dutt Sharma, N., Cavallaro, L., and Bos, H. Memory errors: The past, the present, and the future. In *RAID* (2012), pp. 86–106.