SUDUTA: Script UAF Detection Using Taint Analysis

John Galea(✉) and Mark Vella

PEST Research Lab, University of Malta, Msida, Malta
{john.galea.10,mark.vella}@um.edu.mt

Abstract. Use-after-free (UAF) vulnerabilities are caused by the use of dangling pointers. Their exploitation inside script engine-hosting applications, e.g. web browsers, can even bypass state-of-the-art countermeasures. This work proposes SUDUTA (**S**cript **U**AF **D**etection **U**sing **T**aint **A**nalysis), which aims at facilitating the diagnosis of UAF bugs during vulnerability analysis and improves an existent promising technique based on dynamic taint tracking. Firstly, precise taint analysis rules are presented in this work to clearly specify how SUDUTA manages the taint state. Moreover, it shifts its analysis to on-line, enabling instrumentation code to gain access to the program state of the application. Lastly, it handles the presence of custom memory allocators that are typically utilised in script-hosting applications. Results obtained using a benchmark dataset and vulnerable applications validate these three improvements.

Keywords: Use-after-free · Vulnerability analysis · Taint analysis

1 Introduction

Use-After-Free (UAF) vulnerabilities are memory corruption bugs that pose a serious software security threat. They are caused by the use of dangling pointers, and are particularly targeted inside client-side applications that host script-engines and expose host-application objects to scripts, e.g. web browsers and PDF viewers. Their exploitation can even break state-of-the-art operating system mitigations [11], and result in hijacking control-flow and leaking sensitive information. In just the first three months of 2015, the count of publicly disclosed UAF bugs in script-hosting applications was already up to 20[1]. It is crucial that UAF bugs are found by security researchers before hackers start exploiting them, yet their detection still relies on a predominantly manual procedure during vulnerability analysis. Typically, application binaries are first tested using random inputs, with the intent of crashing the application and discovering potential bugs in an automated manner (fuzzing). Once a crash occurs, manual diagnosis is then carried out by an analyst, utilising assembly debuggers, so that the source of the

The work disclosed is partially funded by the Master it! Scholarship Scheme (Malta).

[1] https://cve.mitre.org.

S. Foresti (Ed.): STM 2015, LNCS 9331, pp. 136–151, 2015.
DOI: 10.1007/978-3-319-24858-5_9

bug can be identified and its exploitation assessed [7]. Tools such as *PageHeap*[2] can be of aid in forcing crashes as close as possible to the bug location, but still do not eliminate manual analysis. Source code-level vulnerability scanning presents an automated option, however, analysing script-hosting applications involves the consideration of a multitude of combinations of code-block executions related to the many ways host application objects can be manipulated by scripts. Consequently, scalability issues arise when such an approach is adopted.

Alternatively, dynamic code analysis can be employed. Similar to fuzzing, it operates upon the executing binary file, with the difference that it aims to automate the detection and diagnosis of UAF vulnerabilities. One promising technique that is set in this direction has been implemented in a tool called Undangle [4]. It follows the fuzzing step by carrying out program information flow analysis of recorded execution traces. Specifically, Undangle carries out dynamic taint analysis [10], where only the 'tainted' flows of interest are marked and analysed, which in this case include the creation, propagation, deletion and dereferencing of pointer data. UAF bugs are immediately detected whenever dangling pointers are dereferenced or not properly cleared. The strength of this approach lies in the fact that UAF bug detection is tackled at its root cause. However, various possibilities for improvement exist. Firstly, the rules that specify taint analysis are ambiguously defined by using natural language, thus hindering both its understandability and reproducibility. Secondly, analysis is performed on instruction traces during a subsequent 'off-line' step, and loses access to the program state. Since some of the taint analysis rules, as well as for report generation, require access to program state, Undangle resorts to instruction emulation. This is more of an indirect solution rather than an appropriate one. Finally, a third limitation entails that all memory management functions, from which most pointer data is introduced, need to be manually defined. Taking into account that many script-hosting applications make use of custom memory allocators for their script engines [2], this limitation complicates UAF detection.

This work builds upon the technique underlying Undangle, aiming to improve both its reproducibility and effective use in the context of script-hosting applications by addressing its limitations. In this work, we propose SUDUTA (**S**cript **U**AF **D**etection **U**sing **T**aint **A**nalysis), which uses taint analysis rules that precisely specify how every x86 instruction updates the program's taint state. It manages the large size of this instruction-set by grouping instructions into equivalent classes. SUDUTA uses Just-In-Time (JIT) binary modification to weave in the code that implements the taint analysis rules directly with the application's execution trace, thus providing access to the program state. This analysis technique is the 'on-line' alternative to Undangle's off-line approach. Additionally, SUDUTA integrates a set of existing heuristics for memory management function identification in order to handle applications with custom memory allocators. Results obtained by experimenting with a benchmark dataset and vulnerable

[2] https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561(v=vs.85).aspx.

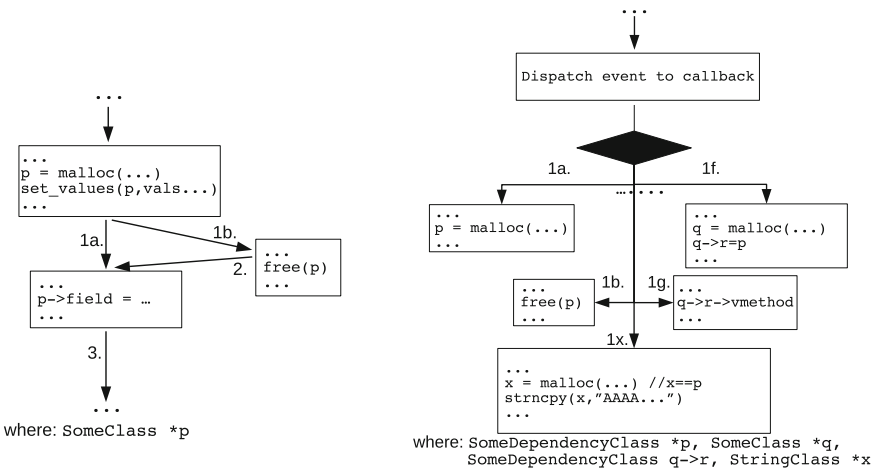
applications show that SUDUTA is an effective UAF detector, and validates the taint analysis rules and its custom memory allocator handling capability. Avenues for optimizing analysis time are also identified.

This work is organized as follows: Sect. 2 expands further on UAF vulnerabilities, Undangle and taint analysis. Section 3 presents SUDUTA, and experimentation and comparison to existing work are detailed in Sects. 4 and 5 respectively. Lastly, Sect. 6 concludes this work.

2 Background

2.1 UAF Vulnerabilities

The control-flow graph (CFG), shown in Fig. 1a, illustrates an example of a simple UAF vulnerability that does not involve a script engine. The erroneous execution sequence takes the path *1b* → *2*, since the dangling pointer `p` is used to access the `field` data of the previously freed memory object. Clearly, similar UAF bugs are trivial to detect via static code analysis, e.g. [6]. However, their complexity pales in comparison with the erroneous sequences involved in script-hosting applications.



(a) UAF (1b → 2). (b) Script UAF (1a → 1f → 1b → 1x → 1g).

Fig. 1. Increased complexity of UAF bugs in script-hosting applications.

In the context of web browsers, such applications have their code arranged in a series of callback functions that are invoked as a result of parsing HTML and JavaScript statements. In turn, these functions manipulate host-application objects, e.g. the DOM tree. Consequently, the execution sequence that triggers an exploitable bug can be quite convoluted, as seen in the CFG shown in Fig. 1b.

The execution of the script-controlled callback sequence *1a* → *1f* results in a memory region being referenced in separate parts of code by the two pointers `p` and `q->r`. On the execution of *1b*, this memory region is freed, and `q->r` becomes a dangling pointer. Furthermore, the execution of *1x* results in a third pointer, `x`, that also references the same region, and has been reused to serve for a new allocation request through `malloc`. Moreover, `strncpy` overwrites the original content that was an object structure for `p`, resulting in the corruption of its virtual function table (vtable) pointer. The block that is reached via *1g* is comprised of the *use* part of the bug, and enables control-flow hijacking when the `q->r->vmethod` virtual method is called through the corrupted vtable pointer that is overwritten with an attacker-controlled value. The root cause of the vulnerability is the dangling pointer creation in the block reached by *1b*. This could have been avoided through correct reference counting, where the memory region is freed only if its reference count reaches 0. However, in such complex scenarios where the same memory object is referenced from disparate code locations, reference counts become increasingly difficult to track correctly. Furthermore, the possibility of executing call-back handling code blocks in various combinations could place UAF detection beyond the reach of static code analysis. In fact, the second scenario is more complex than that shown in Fig. 1a.

2.2 Undangle

The UAF detection method implemented by Undangle [4] is the basis of our work. It detects UAF vulnerabilities by performing dynamic taint analysis [10], a technique which revolves around inspecting data flows of interest. In particular, Undangle is concerned with data flows that create, propagate, dangle, and dereference pointers. Overall, Undangle is a two-step approach. It firstly generates execution traces of the program under analysis from fuzzed inputs, and then proceeds by examining these traces in an off-line fashion. During this second step, Undangle analyses each trace instruction and marks, or *taints*, registers and memory locations that store pointer data. For each tainted location, a taint label stores its 'dangling' status that is maintained through memory management function tracking. Taint propagation occurs when pointer values are copied to other locations or used to derive other pointers via pointer arithmetic. As a result, these new pointers are associated with the same taint information that is linked to their sources. Since Undangle operates off-line, it has to perform instruction emulation to calculate the values of these new pointers. Additionally, a pointer is untainted and no longer analysed when it is overwritten with an untainted value, e.g. `NULL`. Whenever a register/memory address operand is dereferenced, Undangle firstly checks its taint label, and if it is a dangling pointer, a detailed report that includes the bug's location(s) is produced. Undangle uses two maps to store the program's taint state, namely, the *forward map* which links pointers to their corresponding taint labels, and the *reverse map* which associates the start location of a memory object with all the pointers that refer to it. The purpose of the reverse map is to link together all the dispersed pointers associated with the same UAF bug.

The example in Listing 1.1 demonstrates how Undangle tracks taint information and detects UAF bugs. Line 4 is a `ret` instruction that concludes a memory allocation function, which would just have stored a pointer to the allocated region in register `eax` (line 3). As a result `eax` is tainted, indicating that it stores pointer data. Taint propagation occurs in line 6 as the value stored in `eax` is copied to `edi`. This means that `edi` is now also marked as a pointer. At the stage where this region is deallocated by passing this same pointer value to a deallocation routine (lines 8–11), all pointers referring to this region will have their taint status updated to 'dangling'. Assuming `edi` still points to this region, its status would be set to 'dangling' and an alert is raised as soon as it is dereferenced in line 13. By taking an on-line approach, calculating new pointer values, such as that computed in line 15, is done automatically by the processor, and are readily accessible from the program state. However, this does not apply to Undangle's off-line approach, which requires performing instruction emulation to derive pointers and obtain their values.

Listing 1.1. An execution trace under analysis

```
1  call <alloc_function>
2  ...
3  mov eax <@start_of_allocation>
4  ret // Return from alloc function call
5  ...
6  mov edi, eax
7  ...
8  push edi
9  call <dealloc_function>
10 ...
11 ret // Return from dealloc function call
12 ...
13 mov ebx, dword ptr[edi]
14 ...
15 add edi, 07
```

Undangle relies on the analyst to provide information related to memory management functions, which as seen in the previous example, their correct identification is important in detecting UAF bugs. Whilst the functions provided by operating systems (e.g. Window's `VirtuallAlloc()` etc.) or standard run-times (e.g. C's `malloc` etc.) can be easily identified via available documentation, many script-hosting applications employ custom memory allocators to improve performance. Internally, custom memory allocators work by requesting large pools of memory through general purpose allocators, and subsequently use custom functions to manage these buffers in order to handle memory allocation requests [5]. As shown in Fig. 2, if the analyst is unaware of the utilisation of undocumented custom memory allocators and Undangle solely monitors general purpose functions, only the allocation of the larger memory pools would be observed, and not their subdivisions into smaller allocations. Consequently, pointers referring to objects managed by custom memory allocators would not be analysed correctly due to a mismatch in the allocation status between the pool/custom allocator levels (zones A-B and C-D in Fig. 2), thus opening the door to false negatives.

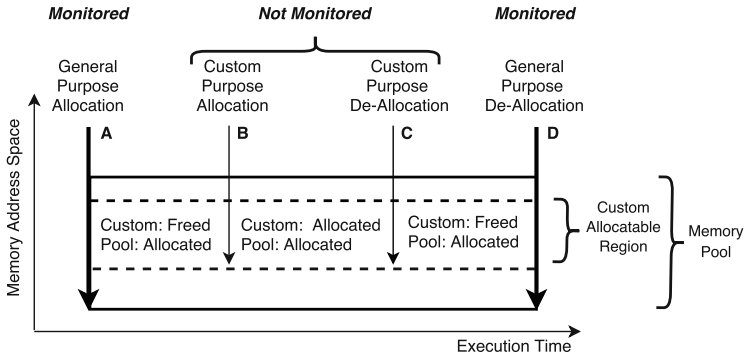


Fig. 2. Pool/custom allocator memory allocation status mismatches

2.3 Formalizing Taint Policy Rules

Another issue with the technique used for Undangle is that the taint analysis rules are only informally described. This aspect lead to replication difficulties due to its ambiguity, and we found the necessity for a notation that precisely defines the taint transitions with respect to the program's own state transitions. Existing notation, previously used to define the taint operation semantics for an intermediate representation language (lifted from assembly) also in a dynamic code analysis setting [10], acts as a basis for this work. However, in our case, we do not lift assembly to an intermediate representation as we define taint rules directly over machine instructions and the machine's context (CPU registers and the virtual memory address space) using the following format:

General Taint Policy Rule:

$$\frac{\text{computation}}{\langle\text{machine context}\rangle\langle\text{taint state}\rangle\text{instruction} \rightsquigarrow \langle\text{taint state'}\rangle}$$

Rules are read from bottom to top and left to right. The bottom-left part of the rule identifies the applicable instruction. Upon an instruction match, the computation defined in the top part of the rule updates the current *taint state* (bottom-left) to the new *taint state'* (bottom-right) as part of a taint state transition ($\rightsquigarrow$). While the machine context (the program state) maps each register/memory address to a q/dword value, its taint state maps each register/memory address to a taint label, specifying whether it is un/tainted. Computation of the new taint state may require values from the current machine context (bottom-left), as in the case of an effective memory operand, e.g. `0x120000[ebp + esi*4]`. During taint analysis, for each instruction in the trace (except those that implement memory management functions), pattern matching is carried out with the rules in order to update the program's taint state. Fundamentally, the rule-set must completely cover the relevant instruction-set, as otherwise the program's taint state would only be partially defined.

3 SUDUTA

We now describe SUDUTA, focusing mainly on how it extends Undangle in relation to our contributions, namely: the formal taint policy rules that precisely describe taint state transitions per x86 instruction; on-line dynamic taint analysis using JIT binary modification; and the provision of custom memory allocator monitoring for accurate UAF detection. It is intended to be used by security researchers during vulnerability analysis in order to facilitate the detection and diagnosis of UAF bugs.

3.1 Taint Policy

SUDUTA keeps track of the program taint state by accessing the machine context, and maintaining taint labels and the forward/reverse maps. The machine context access function ($\triangle$) maps register/memory locations (domain M) to their value (range V). The forward map τ maps register/memory locations (domain M) to taint labels if any (range $T \cup \varepsilon$), whilst the reverse map π associates root addresses (domain V) to a list of all registers/locations storing a pointer to that memory region (range $[M]$).

$$\triangle : M \longrightarrow V$$
$$\tau : M \longrightarrow T \cup \varepsilon$$
$$\pi : V \longrightarrow [M]$$

where: type M: {memory_addresses} $\cup$ {registers},
type V: q/dword, type T: {TaintLabel}

Specifically the taint label structure is defined as:

```
type: struct TaintLabel = {
    state:        LIVE | DANGLING;
    dangling_pc:  V;
    root:         V;
    }
```

It identifies whether a pointer is in a live (pointing to an allocated region) or dangling (pointing to a freed region) state, the value of the program counter (pc) when it turned dangling, and its root address (the start address of the memory buffer it points to as returned by a call to a memory allocation function). Updating the image y of x within a map is denoted by $map[x \leftarrow y]$. Whenever the elements of a map's range are lists, appending or removing values to/from the lists are denoted respectively by $map[x \hookleftarrow y]$ and $map[x \hookrightarrow y]$.

For each traced instruction, SUDUTA updates the taint state based on the instruction and its operands. In the case of x86, an operand can be either a register e.g. `eax`, a memory location e.g. `[0x12000]`, or an immediate i.e. a constant value. Taint labels are only associated with registers and memory locations, and immediate operands are not applicable. Accessing the taint labels for memory operands can become complicated when expressed in terms of register values,

e.g. `0x12000[ebp+esi*4]`. In such cases, getting the required taint label firstly requires accessing the register values and evaluating the resultant address. Therefore, access is needed to both the machine context and the taint state, which are available through $\triangle$ and τ respectively. The operation of evaluating an operand *opnd* into its location m and accessing its taint label t within the current machine context/taint state, is expressed as: $\triangle,\ \tau \vdash opnd \Downarrow \langle m,\ t \rangle$. Note, in the case of a register or a fixed memory location operand, evaluation is not necessary.

Rules are specified over instruction equivalence classes, grouped by instructions that trigger the same state transition as shown in Listing 1.2. Operands that serve both as source and destination operands are either represented as separate *src*/*dst* arguments, or as a combined single *srcdst* argument, depending on their suitability with regards to their applicable taint rules.

Listing 1.2. Instruction groupings used by taint policy rules.

```
⊙(call_addr, v, ret) ::= call
where:
– call_addr is an entry point to an allocation function
– v is the returned value pointing to the start of the
    allocated memory
– ret the register/location storing v
⊘(call_addr, v) ::= call
where:
– call_addr is an entry point to a deallocation function
– v points to the memory to be freed
◁(dst, src) ::= mov | movs | rep_movs | push | pop
⊗(srcdst_1, srcdst_2) ::= xchg
◊(dst, src_1, src_2) ::= add | sub | lea
where:
– result is not an overflow/underflow
– src'_2s value is a not a memory address
– in the case of lea: src_1 is the base or index register
□ ::= inc | dec | nop | cmp | test
∘(dst_1, ..., dst_n, src_1, ..., src_k) ::= all else
```

Rule 1 is a taint introduction rule where the invocation of a memory allocation function constitutes a taint source. When encountering a $\odot(call\_addr, v, ret)$ type of instruction, the register/location m_1 that stores the returned heap pointer is evaluated using $\triangle, \tau \vdash ret \Downarrow \langle m_1, t_1 \rangle$, and its taint label t_1, if existent, is also retrieved. If this location contained pointer data, i.e. it has an associated taint label, the overwrite by the return value v implies that all prior taint-related information needs to be cleared from the taint state, and then updated with the new taint information. This entails firstly removing the existing reverse map entry ($\pi'' = \pi[t_1.root \hookrightarrow m_1]$), and then creating a new taint label, indicating that this location is a live pointer to root address v ($t_{live} = \langle$LIVE, NULL, $v\rangle$). This taint label is used to overwrite the existing label or create a new entry in the forward map ($\tau' = \tau[m_1 \leftarrow t_{live}]$), as well as add/update the reverse map entry associated with v ($\pi' = \pi''[t_2.root \hookleftarrow m_1]$).

Rule 1 - Live pointer introduction:

$$\frac{\triangle, \tau \vdash ret \Downarrow \langle m_1, t_1\rangle,\ \pi'' = \pi[t_1.root \hookrightarrow m_1],\quad t_{live}{=}\langle \text{LIVE, NULL, } v\rangle,\ \tau'{=}\tau[m_1{\leftarrow}t_{live}],\ \pi'{=}\pi''[v{\hookrightarrow}m_1]}{\triangle, \tau, \pi, \odot(call\_addr,v,ret) \rightsquigarrow \tau', \pi'}$$

Rules 2–4 follow a similar structure to Rule 1. However, they do not create any new taint labels but associate existing ones with new registers/locations. On the other hand, instructions related to Rule 5 do not update the taint state, whilst Rule 6 is concerned with untainting registers/locations. The latter also covers instances where arithmetic operations result in overflow/underflows or any other kind of invalid pointer values.

Rule 2 - Move propagation:

$$\frac{\triangle, \tau \vdash dst \Downarrow \langle m_1, t_1\rangle,\ \pi'' = \pi[t_1.root \hookrightarrow m_1],\quad \triangle, \tau \vdash src \Downarrow \langle m_2{,}t_2\rangle,\ \tau'{=}\ \tau[m_1{\leftarrow}t_2],\ \pi'{=}\ \pi''[t_2.root{\leftarrow}m_1]}{\triangle, \tau, \pi, \lhd(dst,src) \rightsquigarrow \tau', \pi'}$$

Rule 3 - Pointer arithmetic propagation:

$$\frac{\triangle, \tau \vdash dst \Downarrow \langle m_1, t_1\rangle,\ \pi'' = \pi[t_1.root \hookrightarrow m_1],\quad \triangle, \tau \vdash src_1 \Downarrow \langle m_2{,}t_2\rangle,\ \tau'{=}\ \tau[m_1{\leftarrow}t_2],\ \pi'{=}\ \pi''[t_2.root{\leftarrow}m_1]}{\triangle, \tau, \pi, \Diamond(dst,src_1,src_2) \rightsquigarrow \tau', \pi'}$$

Rule 4 - Exchange propagation:

$$\frac{\triangle, \tau \vdash srcdst_1 \Downarrow \langle m_1, t_1\rangle,\ \triangle, \tau \vdash srcdst_2 \Downarrow \langle m_2, t_2\rangle,\ \tau' = \tau[m_1 \leftarrow t_2,\ m_2 \leftarrow t_1],\quad \pi'{=}\ \pi[t_2.root{\hookrightarrow}m_2,\ t_2.root{\leftarrow}m_1,\ t_1.root{\hookrightarrow}m_1,\ t_1.root{\leftarrow}m_2]}{\triangle, \tau, \pi, \otimes(srcdst_1,srcdst_2) \rightsquigarrow \tau', \pi'}$$

Rule 5 - No operation propagation:

$$\frac{\tau' = \tau,\ \pi' = \pi}{\triangle, \tau, \pi, \square \rightsquigarrow \tau', \pi'}$$

Rule 6 - Untaint:

$$\frac{\triangle,\ \tau\ \vdash\ dst_1..dst_n\ \Downarrow\ \langle m_1, t_1\rangle..\langle m_n, t_n\rangle,\quad \tau'{=}\ \tau[m_1{\leftarrow}\varepsilon]..\tau[m_n{\leftarrow}\varepsilon],\ \pi'{=}\ \pi[t_1.root{\hookrightarrow}m_1]..\pi[t_n.root{\hookrightarrow}m_n]}{\triangle, \tau, \pi, \circ(dst_1..dst_n,src_1..src_k) \rightsquigarrow \tau', \pi'}$$

Rule 7 relates to dangling pointer creation, which occurs whenever a deallocation function is called. In such cases, all pointers referring to the deallocated memory region starting at v (retrieved through $\pi(v)$) are assigned to the taint label $t_{dangling}$, that associates the register/locations with a dangling state, and records the responsible code location ($\triangle(pc)$).

Rule 7 - Dangling pointer creation:

$$\frac{t_{dangling} = \langle \text{DANGLING}, \triangle(pc), v\rangle,\ \pi(v) = [m_1, .., m_n],\quad \tau' = \tau[m_1{\leftarrow}t_{dangling}]..\tau[m_n{\leftarrow}t_{dangling}],\ \pi' = \pi}{\triangle, \tau, \pi\ \oslash(call\_addr,v) \rightsquigarrow \tau', \pi'}$$

UAF Detection. Having specified, via formal taint rules, how taint state is maintained, we now move on by explaining how SUDUTA detects UAF vulnerabilities (i.e. dereferences of dangling pointers). For every instruction monitored, SUDUTA calls the `UAF_Check` function, which, at a high-level, is described in algorithm 1. At the machine code level, pointer dereferences occur as memory operands that are computed out of register values, termed 'effective'. The operand could simply constitute a single register e.g. `[ebx]`, or could consist of more components e.g. `[ebx +esi*4]`. Importantly, the fact that `ebx` contains pointer data implies that both these example are cases of pointer dereferencing. More specifically, if `ebx` is pointing to the start of an allocated buffer on the heap, `[ebx]` retrieves the first value in this region, whilst `[ebx +esi*4]` obtains a value from an offset, e.g. as in the case of accessing a value from an array. Therefore, this function involves accessing and retrieving the taint labels for all individual registers inside source and destination operands. It assumes the existence of a function `Get_Regs_From_Effective` that returns all registers of an effective operand. An alert is raised if any of them are in a dangling state.

Algorithm 1. `The UAF_Check` function

```
   Input: △, τ, π, inst
   Result: Raises an alert if an UAF vulnerability is detected
1  effective_oprnd_list = Get_All_Effective_Oprnds(inst);
2  for all oprnd ∈ effective_oprnd_list do
3  |  reg_list = Get_Regs_From_Effective(opnd);
4  |  for all reg ∈ reg_list do
5  |  |  if τ(reg) == ⟨DANGLING, *, *⟩ then
6  |  |  |  Raise_Alert(△, τ, π, inst);
7  |  |  end
8  |  end
9  end
```

3.2 On-line Dynamic Taint Analysis

SUDUTA achieves its on-line taint analysis capabilities by carrying out Just-In-Time (JIT) binary modification of the analysed application. As shown in Fig. 3, instead of immediately executing application code, it is firstly copied, per-code block, into an intermediary code cache on the fly. Essentially, the cache provides the means to modify code at an individual instruction granularity. Specifically, SUDUTA inserts a number of 'transparent' calls prior to trace instructions, and thus they do not interfere with the state of the analysed application. This approach requires that CPU registers are duly saved and restored before/after these calls. Also, the code to which control flow is transferred, as well as the code driving the code-cache mechanism, must not interfere with the application's data. These calls invoke functions that encode SUDUTA's taint policy and that

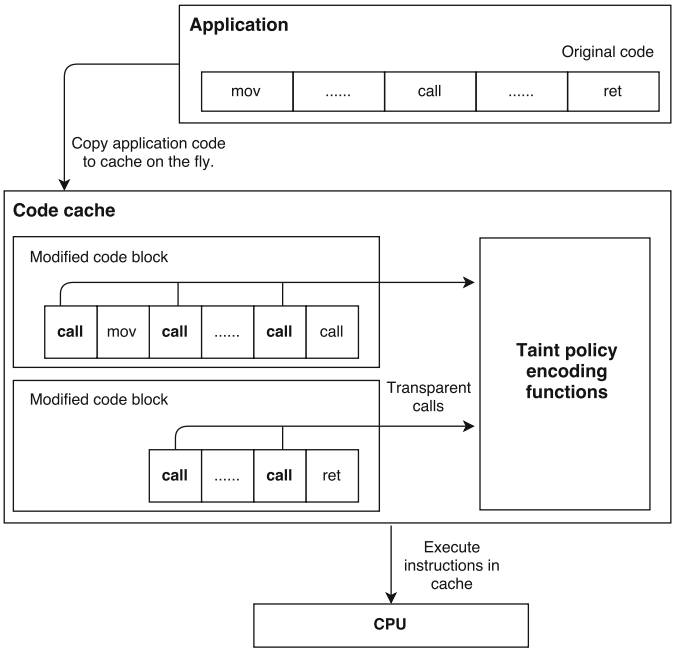


Fig. 3. On-line dynamic taint analysis using JIT binary modification.

have access to the entire machine context, enabling on-line dynamic taint analysis. The encoding functions also keep track of separate machine contexts per application thread. This is only necessary for CPU registers, since taint labels for memory locations are not thread-specific. Finally, each individual code block is executed only after binary modification is complete.

Additional noteworthy aspects of SUDUTA concern user/kernel-mode switching, optimized untainting, and selective module tracing. Since such applications do not usually comprise kernel-level components, on-line taint analysis is optimized by restricting traces to user-mode instructions. Essentially, kernel-mode instructions are ignored and instruction tracing resumes with the first user-mode instruction following a system call. However, this approach introduces intricate situations where kernel code updates the application's memory, and untaint should occur in event that a pointer's value is overwritten. SUDUTA takes the approach of keeping shadow values for all tainted locations, which are consulted prior to pointer propagation, dangling pointer creation and pointer dereference checks. When a current value does not match its corresponding shadow value, the untaint rule (Rule 6) is instead applied. The same method is in fact utilized as a general optimization strategy for untainting. Considering the large number of instructions applicable to this rule, the optimisation saves on the number of control flow transfers to its encoding function by performing untaint in the other rule-encoding functions when they detect a value/shadow value mismatch.

One final optimization involves selective module tracing. Similar to switching off tracing for kernel-mode code, it might be desirable that analysis is also switched off whenever execution control is transferred to loaded libraries that are not of interest. For example, in the case of a web browser, a system library that handles GUI window management and that excludes any form of memory management, is neither expected to contain an application-specific UAF bug nor affect the detection of UAF bugs inside prominent modules. Similarly, libraries that do not contain script/DOM-related code may be considered low priority and their exclusion might speed-up analysis time. Pointer modifications performed by the excluded modules are handled via the aforementioned optimized implementation of the untaint rule.

3.3 Custom Memory Allocator Monitoring

SUDUTA aims to maximize UAF detection by identifying undocumented custom memory allocators in a preliminary stage. It makes use of existing heuristics [5] that capture the typical characteristics of memory management functions. They are particularly suitable since they are independent of the data structures used for memory pool management. SUDUTA implements the heuristics in a filter-based approach, where all executed functions, except for provided memory management functions, are initially considered as candidates. Those that do not adhere to the heuristic-based filters are progressively removed. The memory allocation heuristics are: *(H1)* The function should return a heap pointer, *(H2)* Returned heap pointers should be used to firstly initialise memory before they are used for reading, and *(H3)* The function should not return the same heap address twice unless previously freed. The memory deallocation heuristics are: *(H4)* The function should take as a parameter a memory address previously returned by an allocation function, *(H5)* The function should reference a common memory address region shared with custom allocation functions, which is used for memory pool meta-data, and *(H6)* A freed region can be re-allocated again by custom allocation functions. Although no heuristics specific for memory re-allocation functions have yet been integrated into SUDUTA, their allocation and deallocation sub-components may still be detected by these heuristics.

4 Evaluation

SUDUTA was evaluated using a 32-bit prototype that uses DynamoRIO[3] [3] to implement JIT binary modification and program memory analysis. It consists of 5,657 lines of C code. Experimentation focused on validating its UAF detection capabilities based on taint analysis in an on-line setting, along with its support of monitoring custom memory allocators. Finally, SUDUTA's practicality in terms of analysis time was also explored. All experiments were carried out on a guest OS running Windows XP SP3, with an Intel Core i7 3.2 GHz Quad Core Processor. The reason for choosing this OS was due to availability of working exploits.

[3] http://www.dynamorio.org/.

Validating UAF detection capabilities. SUDUTA's detection capabilities were firstly validated using the Juliet for C\C++ v1.2 vulnerability benchmark test suite. All 459 UAF bugs were detected by SUDUTA without generating any false positives. Moreover, SUDUTA generated no false positives when it analysed the 459 benign test cases in the same test suite. Once basic UAF detection capabilities were confirmed, focus was shifted to script-hosting applications that contain known exploitable UAF vulnerabilities.

We chose three case studies, on the basis of a working exploit and a technical analysis report being available, which allowed us to verify whether the vulnerabilities reported by SUDUTA corresponded to the actual bugs. These were: Internet Explorer 6.0.2900 (CVE-2010-0249), Firefox 3.5.1 (CVE-2011-0073), and MS Excel 2003 (OSVDB-76840). Due to the size of the applications, we made use of SUDUTA's selective module tracing feature to focus on the priority modules that contain callback functions invoked by script engines, as per example shown in Fig. 1b, namely: `mshtml.dll` (IE), `xul.dll` (Firefox), and `vbe6.dll` (Excel). During the first vulnerability analysis, the *Bf3* browser fuzzer[4] was utilised for input generation. However, the fuzzer was not capable of triggering the execution paths affected by the UAF bugs. Therefore, we resorted to using their exploits[5]. Table 1 shows the results obtained. Each exploit was executed twice with the identification and monitoring of customer memory allocators enabled/disabled.

SUDUTA fails to identify the vulnerability in IE 6 without monitoring custom allocators, but is successful when this feature is turned on. This outcome demonstrates the importance of monitoring custom allocators when analysing script-hosting application. In fact, we confirmed through the `mshtml.dll`'s symbol file that this library uses the undocumented custom `MemAllocClear` and `MemFree` functions. This contrasts with Excel's case where monitoring the operating system's `HeapAlloc` and `HeapFree` suffices. In all cases, SUDUTA returned no false positives. Firefox's case was problematic since after 14 h, analysis was still running without having yet detected any vulnerabilities. The cause for this long analysis time was not immediately clear. However, `xul.dll`'s larger size

Table 1. Vulnerability detection results.

ID	Application	Custom allocator monitor	Detected?
CVE-2010-0249	Internet Explorer 6	Disabled	No
		Enabled	Yes
OSVDB-76840	MS Excel 2003	Disable	Yes
		Enabled	Yes
CVE-2011-0073	Firefox 3.5.1	Disabled	No
		Enabled	No

[4] http://www.aldeid.com/wiki/Bf3.
[5] Retrieved from Exploit-DB: https://www.exploit-db.com/.

Table 2. Performance results with respect to analysis time.

ID	Optimised?	Analysis time	Speed-up	Detected?
CVE-2010-0249	No	39.3 s	x0	Yes
	Yes	24.7 s	x1.6	Yes
OSVDB-76840	No	24.5 s	x0	Yes
	Yes	14.8s	x1.6	Yes
CVE-2011-0073	No	$\sim$ 14 h	x0	No
	Yes	1,868.2 s	$\sim$ x27.0	Yes

(x3.51 the size of `mshtml.dll`) pointed towards a trace size issue. This indication forced us to seek possibilities of optimization, which are presented as part of the following discussion on performance results.

Performance. The major bottleneck to analysis time is caused by the fine grained examination of instructions as described by SUDUTA's taint policy. The need to check pointer dereferences at every instruction is particularly expensive. Consequently, an optimisation is adopted that trades off pointer-tracking precision for performance by inspecting a smaller restricted group of instructions, with the aim of reducing analysis time. More specifically, the optimisation only examines the $\lhd$ instruction group and the `lea` instruction in order to propagate taint. Furthermore, instead of considering all instructions, the optimisation solely checks dereferences of dangling pointers operated by `mov` instructions. Consequently, instructions including `add`, `sub`, `inc`, `dec` and `xchg` are not monitored. We base this optimisation on our assumption that host application objects, exposed to scripts, are accessed by utilising the root address, without performing pointer arithmetic. However, this assumption holds in cases where memory objects are not stored contiguously inside an array. Table 2 shows performance results obtained, comparing SUDUTA with its optimised version.

The optimised version of SUDUTA managed to significantly reduce the analysis time. Despite the imprecision incurred due to the monitoring of less instructions, it still detected the vulnerabilities in all applications. No false positives were produced by SUDUTA. With regards to Firefox, the optimised version speeded up analysis by a factor of 27, which resulted in reasonable time to carry out analysis in full and identify the vulnerability. Moreover, the optimised version also detected all UAF bugs present in the Juliet benchmark database, without generating any false positives. In general, SUDUTA's automated procedure lessens the manual effort required to diagnose and remove UAF vulnerabilities.

5 Related Work

Like SUDUTA, several proposed techniques detect UAF vulnerabilities with the overall aim of facilitating vulnerability analysis. Some [6,13] take a static approach but face difficult scalability challenges, such as conducting accurate

point-to analysis, that make them only suitable to analyse small programs. Other techniques tackle the problem dynamically. Conventional memory debugging tools (e.g. Dr. Memory[6]) detect UAF bugs by checking whether dereferenced pointers access memory marked as live, but are ineffective when a used dangling pointer refers to re-allocated memory. Rather than taking a memory-centric approach, SUDUTA adopts a pointer-centric approach and thus avoids this concern.

Instead of detecting UAF vulnerabilities as part of a debugging effort, other works [9,12] focus on inserting dynamic checks during compilation, with the purpose of hardening applications for deployment. They are concerned with minimising overheads, particularly due to their requirement to perform potentially expensive checks upon all pointer dereferences. Recent works [8,14] avoid this bottleneck by nullifying all dangling pointers immediately after the deallocation of their referenced object. Other techniques such as hardened memory allocators [1] or process address spaces (e.g. EMET[7]) aim at mitigating exploits, regardless of the type of vulnerabilities leveraged.

6 Conclusion

UAF vulnerabilities stem from the incorrect dereference of dangling pointers and pose a threat to the security of script-hosting applications. Vulnerability analysts attempt to manually diagnose security holes, and this requires significant effort. Consequently, a need exists for automated tools that facilitate analysis in order to be competitive against adversaries.

In this work, we propose SUDUTA, which builds upon Undangle to address several limitations. Firstly, SUDUTA shifts analysis to on-line, so that program state can be accessed. Moreover, its specification is also defined as a formal taint policy, thus rendering the technique easier to understand and replicate. SUDUTA improves further by also identifying undocumented custom memory allocators automatically, in order to increase detection coverage. Experimentation results validate the approach, particularly the precise taint policy, since UAF vulnerabilities found in benchmark test cases and real-world script-hosting applications were successfully detected. Through the identification of custom memory allocators, SUDUTA manages to detect the vulnerability in IE 6 only when monitoring undocumented memory management functions. Furthermore, trading off pointer-tracking precision for performance improved analysis time greatly, with an average speed-up of x10.1, without producing any false negatives. Results also highlight the need for a smart fuzzing approach. Future work entails designing and integrating an improved fuzzer, where test case generation is based on exploit patterns rather than only grammatically correct syntax. Additionally, upgrading SUDUTA to support 64-bit applications would enable a larger scale evaluation.

[6] http://www.drmemory.org/.
[7] https://support.microsoft.com/en-us/kb/2458544.

References

1. Akritidis, P.: Cling: a memory allocator to mitigate dangling pointers. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security 2010, p. 12. USENIX Association, Berkeley (2010)
2. Argyroudis, P., Karamitas, C.: Exploiting the Jemalloc Memory Allocator: Owning Firefox's Heap. Blackhat USA (2012)
3. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, VEE 2012, pp. 133–144. ACM, New York (2012)
4. Caballero, J., Grieco, G., Marron, M., Nappa, A.: Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Heimdahl, M.P.E., Su, Z. (eds.) ISSTA, pp. 133–143. ACM (2012)
5. Chen, X., Slowinska, A., Bos, H.: Who allocated my memory? detecting custom memory allocators in C binaries. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 22–31 (2013)
6. Josselin, F., Laurent, M., Marie-Laure, P.: Statically detecting use after free on binary code. In: GreHack, pp. 61–71 (2013)
7. Kratzer, J.: Root cause analysis Memory Corruption Vulnerabilities. https://www.corelan.be/index.php/2013/02/26/root-cause-analysis-memory-corruption-vulnerabilities/. Accessed 15 June 2015
8. Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., Lee, W.: Preventing use-after-free with dangling pointers nullification. In: Proceedings of the 2015 Annual Network and Distributed System Security Symposium (2015)
9. Nagarakatte, S., Zhao, J., Martin, M.M.K., Zdancewic, S.: CETS: compiler enforced temporal safety for C. In: Vitek, J., Lea, D. (eds.) ISMM, pp. 31–40. ACM (2010)
10. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 317–331. IEEE Computer Society, Washington, DC (2010)
11. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP 2013, pp. 574–588. IEEE Computer Society, Washington, DC (2013)
12. Xu, W., DuVarney, D.C., Sekar, R.: An efficient and backwards-compatible transformation to ensure memory safety of C programs. ACM SIGSOFT Softw. Eng. Notes **29**(6), 117–126 (2004)
13. Ye, J., Zhang, C., Han, X.: POSTER: UAFChecker: scalable static detection of use-after-free vulnerabilities. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, pp. 1529–1531. ACM, New York (2014)
14. Younan, Y.: Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers (2015)