Grenoble INP **UGA**
Master CybserSecurity **Year 2016-17**

Software Security & Secure Programming

Written Exam - Tuesday January the 24th, 2017

Duration: 3 hours – **all documents allowed** – Answers can be written either in English or in French

This exam contains two distinct parts:

1. Three independent exercises, supposed to be solved in about 30 minutes each;
2. Some questions on a research paper, allow about 1h30 to read the paper and answer these questions.

Exercise 1. ($\sim$ 4 pts)

We consider the following C program:

```
int *f1() {
    int a ;
    a = 42 ;   // (1)
    return &a ;
}

int f2() {
    int b ;
    b = 12 ;   // (2)
    return b ;
}

int main() {
    int *p ;
    p=f1() ;
    printf("%d\n", *p) ;   // (3)
    f2() ;
    printf("%d\n", *p) ;   // (4)
    return 0 ;
}
```

Q1. Draw the program execution stack at two execution points:

1. when function `f1` is executed;
2. when function `f2` is executed;

What are the values printed when this program is ran ?

Q2. Clearly, this program follows a dangerous "programming pattern" from a security point of view regarding the (implicit) relationships between variable `a`, `b` and `p`. Express why in a few words.

Q3. By **sketching** three examples of vulnerable code based on this same programming pattern, explain how this vulnerability would allow an attacker:

1. to disclose some "confidential" data;
2. to alter some "sensible" data;
3. to execute "unexpected code"[1].

These examples should be obtained essentially by replacing instructions noted `(1)`, `(2)`, `(3)` and `(4)` by other instructions of your choice, but you can also change the variable types or add some new variables. Note that you don't need to write complete C code, nor to strictly follow the C syntax.

Q4. State (in few words) a general "C secure programming rule" to warn developers about this insecure programming pattern. Do you think it would be possible to detect such an incorrect pattern statically (i.e., at compile time) ? Explain why – or why not – in a few lines.

Exercise 2. ($\sim$ 4 pts)

We consider the following C function:

```
void foo() {
  int x ;
  int buff[15] ;
  x = 1 ;
  while (x < 12) {
       x = x + 3 ;      \\    (1)
  } ;
  buf[x] = 0 ;
}
```

Q1. Insert the assertions required to make this code "secure" with respect to *arithmetic overflows* and *out-of-bound buffer accesses*.

Is there an execution trace which violates one of these assertions (when executing this code) ?

Q2. Draw the control-flow graph (CFG) of this function (**without** the assertions)

Q3. Give the results obtained when performing a value-set-analysis (VSA) in which the abstract values of variable `x` are expressed using an interval. You should indicate the abstract value of `x` at entry and exit points of each basic block of the function CFG. You are free to use – or not to use – *widening* and *narrowing* operators to perform this VSA (the result would be similar on this example).

Q4. Are the abstract values obtained at question **Q3** able to discharge the assertions ?

Q5. Reconsider questions **Q3** and **Q4** when instruction **(1)** is replaced by

```
x = x * 3 ;
```

What do you conclude ?

[1] e.g., using pointer to functions

Exercise 3. ($\sim$ 4 pts)

We consider the following C function:

```
int foo() {
  int x, y  ;       // input variables
  int z, t ;       // auxilliary variables
  int buff[24] ;

  scanf("%d", &x) ;
  scanf("%d", &y) ;

  if (x < 15) {
      z = x + 4 ;
      if (z < y)
          buff[z] = 0 ;
      else {
          t = x + y ;
      }
  } else {
      t = y * 2 ;
      if (t * x) > 12
     y = y+1 ;
      if (y > x)
         buff[250] = 0 ;
  } ;
return 0 ;
}
```

Q1. Draw the control-flow graph (CFG) of the function.

Q2. Give the path predicate associated to each path in the CFG.

Indications:

- This function CFG contains **six** paths (going from the entry block to the exit block).
- The path predicates are logical propositions over free variables `x0` and `y0` (corresponding to program inputs). A valuation of `(x0, y0)` satisfies a given path predicate if and only if it allows to execute its corresponding CFG path.

Q3. Give a solution (if any !) to each path constraint.

Q4. We would like to **prove**, using symbolic execution, that no out-of-bounds buffer access may occur in this program.

1. How should we strengthen the previous path predicates to express this property ?
2. What is the result obtained on this particular example (i.e., can we prove the absence of out-of-bounds buffer access) ?
3. Is a symbolic execution technique able prove such a property on any program (explain your answer) ?

Questions on a research paper. (∼ 8 pts)

Read the (part of the) paper given in appendix, from the beginning to the end of section 3, answering the following questions.

Q1 [section 1].

What are the software vulnerabilities targeted by the tool Undangle ?
Give an example (for instance in C code) of such a vulnerability.

Q2 [section 1].

Would you classify Undangle as a static analysis tool or as dynamic analysis tool ?

Q3 [section 2].

Illustrate the following terms using **concrete** "C like" examples
(i.e., not by re-phrasing the paper definitions):

- dangling pointer
- unsafe dangling pointer
- sharing and non-sharing dangling pointer bugs
- root pointer

Q4 [section 3].

What is called *early detection* of dangling pointers, and why is it important according to the authors ?

Q5 [section 3].

Explain what is called *pointer tracking* in the paper, and why is is used for.

Q6 [all sections].

Regarding the vulnerability detection technique supported by Undangle:

- is it sound (i.e., no false negatives) ?
- is it complete (i.e., no false positives) ?

Explain your answers ...

Q7 [section 3].

Is Undangle able to detect the vulnerability illustrated in Exercise 1 ?

Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities

Juan Caballero, Gustavo Grieco, Mark Marron, Antonio Nappa
IMDEA Software Institute
Madrid, Spain
{juan.caballero, gustavo.grieco, mark.marron, antonio nappa}@imdea.org

Abstract

Use-after-free vulnerabilities are rapidly growing in popularity, especially for exploiting web browsers. Use-after-free (and double-free) vulnerabilities are caused by a program operating on a dangling pointer. In this work we propose *early detection*, a novel runtime approach for finding and diagnosing use-after-free and double-free vulnerabilities. While previous work focuses on the creation of the vulnerability (i.e., the use of a dangling pointer), early detection shifts the focus to the creation of the dangling pointer(s) at the root of the vulnerability.

Early detection increases the effectiveness of testing by identifying unsafe dangling pointers in executions where they are created but not used. It also accelerates vulnerability analysis and minimizes the risk of incomplete fixes, by automatically collecting information about all dangling pointers involved in the vulnerability. We implement our early detection technique in a tool called *Undangle*. We evaluate Undangle for vulnerability analysis on 8 real-world vulnerabilities. The analysis uncovers that two separate vulnerabilities in Firefox had a common root cause and that their patches did not completely fix the underlying bug. We also evaluate Undangle for testing on the Firefox web browser identifying a potential vulnerability.

1 Introduction

A dangling pointer is created when the object a pointer points-to is deallocated, leaving the pointer pointing to dead memory, which may be later reallocated or overwritten. Dangling pointers critically impact program correctness and security because they open the door to *use-after-free* and *double-free* vulnerabilities, two important classes of vulnerabilities where a program operates on memory through a dangling pointer.

Use-after-free and double-free vulnerabilities are exploitable [14, 31] and are as dangerous as other, better known, classes of vulnerabilities such as buffer and integer overflows. Use-after-free vulnerabilities are particularly insidious: they have been used to launch a number of zero-day attacks, including the Aurora attack on Google's and Adobe's corporate network [52], and another 3 zero-day attacks on Internet Explorer within the last year [10, 48, 53].

Our analysis of the publicly disclosed use-after-free and double-free vulnerabilities in the CVE database [13] reveals two disturbing trends illustrated in Figure 1: (1) the popularity of use-after-free vulnerabilities is rapidly growing, with their number more than doubling every year since 2008 (over the same period the total number of vulnerabilities reported each year has actually been decreasing), and (2) use-after-free and double-free vulnerabilities abound in web browsers (69%) and operating systems (21%), which use complex data structures and are written in languages requiring manual memory management (e.g., C/C++).

Use-after-free and double-free vulnerabilities are difficult to identify and time-consuming to diagnose because they involve two separate program events that may happen far apart in time: the *creation* of the dangling pointer and its *use* (dereference or double-free). In addition, understanding the root cause may require reasoning about multiple objects in memory. While some dangling pointers are created by forgetting to nullify the pointer used to free an object (*non-sharing* bugs), others involve multiple objects sharing an object that is deallocated (*sharing* bugs).

Sharing bugs happen because not all parent objects know about the child deallocation. They are particularly problematic for web browsers, which are built from components using different memory management methods. For example, in Firefox, JavaScript objects are garbage-

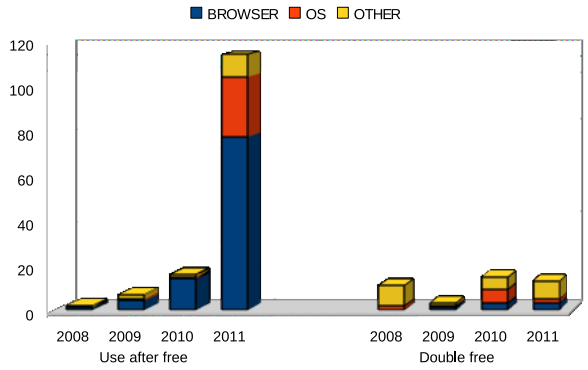

Figure 1: Number of use-after-free (left) and double-free (right) vulnerabilities reported in the CVE database in the 2008-2011 period, split by vulnerabilities in browsers, OSes, and other programs.

collected, XPCOM objects are reference-counted, and the layout engine uses manual management. This mixture makes it extremely difficult to reason about objects shared between code using different memory management methods, which are particularly susceptible to dangling pointers bugs.

Previous work on tools for identifying and diagnosing use-after-free and double-free vulnerabilities [11, 36, 47] and on techniques to protect against their exploitation [15, 19, 29, 30, 46] focus on the use of the dangling pointer, which creates the vulnerability. In this work, we propose a novel dynamic analysis approach for analyzing and protecting against use-after-free and double-free vulnerabilities. Our approach shifts the focus from the creation of the vulnerability (i.e., the dangling pointer use) to the creation of the dangling pointers at the root of the vulnerability. We call our approach *early detection* because it identifies dangling pointers when they are created, before they are used by the program. Early detection is useful for different applications that target use-after-free and double-free vulnerabilities. In this work we evaluate early detection for testing for unsafe dangling pointers and for vulnerability analysis.

Testing for unsafe dangling pointers. A dangling pointer is *unsafe* if it is used in at least one program path and *latent* if the program never uses it. Use-after-free and double-free vulnerabilities are difficult to detect during testing because in a given execution the unsafe dangling pointer may not be created or it may be created but not used. Coverage can be increased using automated input generation techniques [26, 33, 41]. However, if the input space is large it can take long a time to find an input that creates the dangling pointer *and* triggers the vulnerability. Early detection extends the effectiveness of testing by also detecting unsafe dangling pointers in executions where they are created but not used. To identify at runtime unsafe dangling pointers and minimize false positives, we use the intuition that *long-lived* dangling pointers are typically unsafe. Moreover, long-lived dangling pointers are always dangerous and should be removed, even if currently not used, because modifications to the code by the (unaware) programmer may result in new code paths that use the (dangling) pointer. To identify long-lived dangling pointers, early detection tracks the created dangling pointers through time, flagging only those dangling pointers that are still alive after a predefined window of time.

Vulnerability analysis. A common debugging task is, given an input that causes a crash or exploitation, determining how to best patch the vulnerability, which requires understanding the vulnerability type and its root cause. Such crashing inputs are typically found using automatic testing techniques [26, 33, 41] and are usually included in vulnerability disclosures to program vendors. Our early detection technique automatically determines if a crash is caused by a use-after-free or double-free vulnerability and collects diagnosis information about the program state at both creation and use time. State-of-the-art memory debugging tools [11, 36, 47] provide information about the program state when the dangling pointer is used, but provide scant information about the dangling pointer creation (limited to the deallocation that caused it, if at all). This is problematic because a patch needs to completely eliminate the dangling pointer. If a patch only prevents the dangling pointer use that causes the crash, it may be incomplete since a dangling pointer may be used at different program points and there may be multiple dangling pointers created in the same bug. Furthermore, current tools offer little help when debugging sharing bugs, as these bugs require reasoning about multiple objects that point to the deallocated object at creation time. Our early detection technique tracks all pointers that point-to a buffer, automatically identifying all dangling pointers to the deallocated buffer, not only the one that produces the crash. Thus, it offers a complete picture about the type of dangling pointer bug and its root cause.

We implement our early detection technique in a tool called *Undangle* that works on binary programs and does not require access to the program's source code. However, if program symbols are available, the results are augmented with the symbol information.

We evaluate Undangle for vulnerability analysis and testing for unsafe dangling pointers. First, we use it to diagnose 8 vulnerabilities in real-world programs including four popular web browser families (IE7, IE8, Firefox,

Safari) and the Apache web server. Undangle produces no false negatives and uncovers that two use-after-free vulnerabilities in Firefox were caused by the same dangling pointer bug. The reporter of the vulnerabilities and the programmers that fixed them missed this, leaving the patched versions vulnerable to variants of the attacks. We identify this issue with no prior knowledge of the Firefox code base, which shows the value of early detection for diagnosing the root cause of the vulnerability.

Then, we test two recent Firefox versions for unsafe dangling pointers. Early detection identifies 6 unique dangling pointer bugs. One of them is potentially unsafe and we have submitted it to the Mozilla bug tracker. Our disclosure has been accepted as a bug and is pending confirmation on whether it is exploitable. Two other bugs are in a Windows library, so we cannot determine if they are unsafe or latent. The other three bugs are likely latent but our results show that the diagnosis information output by Undangle makes it so easy to understand and fix them, that they should be fixed anyway to close any potential security issues.

This paper makes the following contributions:

- We propose early detection, a novel approach for finding and diagnosing use-after-free and double-free vulnerabilities. Early detection shifts the focus from the creation of the vulnerability to the creation of the dangling pointers at the root of the vulnerability.
- We have designed an early detection technique that identifies dangling pointers when they are created. It uncovers unsafe dangling pointers in executions where they are created but not used, increasing the effectiveness of testing. When diagnosing a crash caused by a dangling pointer, it collects extensive diagnosis information about the dangling pointer creation and its use, enabling efficient vulnerability analysis.
- We have implemented our early detection technique into a tool called Undangle that works directly on binary programs. We have evaluated Undangle for vulnerability analysis using 8 real-world vulnerabilities and for testing on two recent versions of the Firefox web browser.

2 Problem Overview

Dangling pointers are at the root of use-after-free and double-free vulnerabilities. Both classes of errors involve two events that may happen far apart in time: the creation of the dangling pointer and its use. We illustrate them in Figure 2. First, the program creates one or more dangling pointers by deallocating a memory object (Object 1 at t_{create}). The dangling pointers live until they are destroyed by modifying their value (set to NULL at $t_{destroy}$ in Figure 2) or deallocating the memory where they are stored. If the program uses a dangling pointer before destroying it (dereferenced or double-freed at t_{use}), it operates on unknown content since the memory pointed-to is dead and may have been re-allocated to a different object (Object 2 at $t_{memreuse}$).

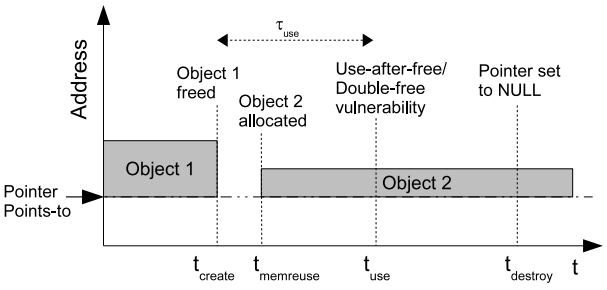

Figure 2: Dangling pointer lifetime.

A use-after-free vulnerability happens when the program dereferences a dangling pointer. An attacker can exploit a use-after-free vulnerability for reading secret values, overwriting sensitive data, and for control-flow hijacking. A common hijacking technique is to heap spray malicious content to overwrite a function pointer or an object's virtual table address in the range pointed-to by the dangling pointer [14]. A double-free vulnerability happens when the program passes a dangling pointer as a parameter to a deallocation, freeing the (already free) memory range it points-to. This may corrupt the internal heap metadata enabling an attacker to exploit the program [31].

In the remainder of this section we detail early detection for identifying unsafe dangling pointers (Section 2.1) and for vulnerability analysis (Section 2.2); provide useful definitions (Section 2.3); and present an architectural overview of Undangle (Section 2.4).

2.1 Identifying Unsafe Dangling Pointers

Determining that a dangling pointer is never used (i.e., it is latent) requires a precise points-to and reachability analysis along all inter-procedural paths between creation and destruction. Such detection on large programs like web browsers is beyond state-of-the-art static analysis tools. Instead, current runtime tools determine that a dangling pointer is unsafe if it happens to be used in an execution. This late detection prevents the identification of dangling pointers that are created but not used in the monitored execution.

To identify unsafe dangling pointers at runtime, even if they are not used, we utilize the intuition that long-lived dangling pointers are likely to be unsafe. Even when only latent, they should be removed because later modifications to the code may make them unsafe. On the other hand, short-lived dangling pointers may have been introduced during compilation, or left temporarily dangling by the programmer. For example, a programmer should always nullify the pointer passed to a deallocation after the deallocation returns (`free(aptr); aptr=NULL;`). This correct behavior still produces some transient dangling pointers because at the binary level `aptr` is copied into the stack before calling the deallocation function. Right after the deallocation returns, there are at least two dangling pointers: the parameter in the stack and `aptr`. However, both will be destroyed in the next few instructions and thus, are not unsafe. The programmer may also introduce transient dangling pointers. For example, when destroying a tree structure, a programmer may create a dangling pointer by deallocating a child node before its parent node, but the dangling pointer is soon destroyed when the parent node is deallocated.

Safety window. To determine when a dangling pointer becomes unsafe, our early detection technique takes as input the size of a *safety window*, given as a number of instructions executed. The safety window size captures when we believe a dangling pointer is no longer short-lived and should be considered unsafe. When a dangling pointer is created, a callback is set for the time of creation plus the safety window size. If the dangling pointer is still alive when the callback triggers, it is flagged as unsafe. Dangling pointers used by the program are flagged regardless if their callback has triggered. Time is measured independently for each thread using a per-thread instruction counter. This way, if a thread creates a dangling pointer and goes to sleep, time freezes for that thread.

The size of the safety window varies depending on the application. For testing, we set the safety window to a small positive value to allow the program to destroy short-lived dangling pointers. We evaluate safety window size selection for testing in Section 4.3, selecting a size of 5,000 instructions.

Coverage. To find unsafe dangling pointers many program paths need to be explored. Our early detection technique requires an external tool to produce inputs that traverse different paths in the program. We run the program on the inputs produced by the input generation tool, and apply early detection to each execution. Currently, we use the bf3 (Browser Fuzzer 3) [1] tool to generate inputs, but many other tools exist that could be used instead [24, 26, 33, 41].

Output information. When a dangling pointer is used or becomes unsafe, early detection outputs detailed information on the program state when the dangling pointer was created and when the dangling pointer was used or became unsafe. This diagnosis information includes: if the dangling pointer was used or flagged as unsafe; if used, the vulnerability type (use-after-free or double-free); a description of the deallocated buffer (e.g., address, size, callsite); which thread deallocated the object and which thread used it; the program callstack; the list of dangling pointers alive; and information about the buffers storing the dangling pointers. We detail the output information in Section 3.3.

2.2 Vulnerability Analysis

While not as general as existing memory debugging tools that detect a wider range of errors [11, 12, 36, 47], Undangle can be used as a specialized diagnosis tool for use-after-free and double-free vulnerabilities Undangle detects all dangling pointer uses, similar to Electric Fence [47] and PageHeap [12], which use a new page for each allocation and rely on page protection mechanisms to detect dangling pointer uses. This is better than popular memory debugging tools such as Valgrind [11] and Purify [36], which check if a dereferenced pointer points-to live memory, missing dangling pointers that, when dereferenced, point to an object that has reused the memory range ($t_{use} \geq t_{memreuse}$ in Figure 2).

The main advantage of using Undangle for vulnerability analysis is that if the crash/exploit is due to a use-after-free or double-free vulnerability, Undangle automatically collects more information about the root cause of the vulnerability than any existing tool. Detailed diagnosis information can be collected because Undangle detects the dangling pointers at creation and tracks them until they are used. Detailed diagnosis information accelerates vulnerability analysis and minimizes the risk of incomplete fixes [35]. To use Undangle for vulnerability analysis we configure it with an infinite safety window size, so that dangling pointers are only flagged when they are used by the program. Undangle offers the following benefits for vulnerability analysis:

- It automatically identifies the dangling pointer creation and outputs detailed information (described in the previous section) about the program state at that time. In comparison, current tools will at most flag which deallocation created the dangling pointer used in the vulnerability (if at all). The analyst still needs to rerun the program in a debugger to obtain information about the program state at creation time,

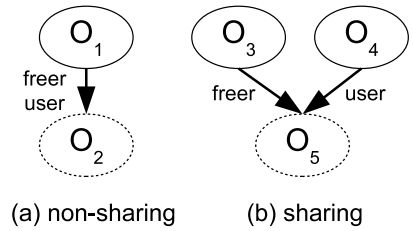

Figure 3: Classes of dangling pointer bugs. The dashed object is deallocated creating the dangling pointers.

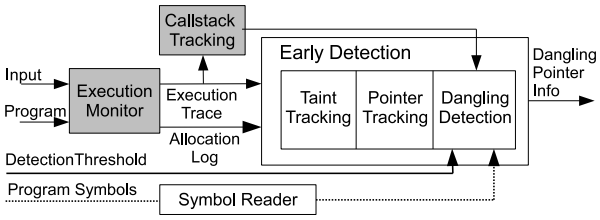

Figure 4: Architecture Overview. Gray boxes were previously available.

which may differ in another execution due to non-determinism in the program (particularly wrt. memory allocator behavior).

- It automatically identifies not only the dangling pointer that causes the crash, but also all other dangling pointers that point-to the deallocated buffer at creation time, as well as the memory objects where they are stored. This is fundamental to reason about sharing bugs, where multiple parent objects hold pointers to the deallocated child object. Sharing bugs are most complex to understand and their fix is not as simple as nullifying the offending pointer, since all parents need to have a way of determining if the child was deallocated. Identifying all dangling pointers created by deallocating a buffer is also fundamental to minimize the risk that a patch is incomplete, since those dangling pointers be also be unsafe.

2.3 Definitions

We define a pointer to be a pair comprising: its *store address*, i.e., the lowest memory address where the pointer is stored, and its *points-to address*, i.e., the pointer's value. A dangling pointer is a special type of pointer. A pointer becomes a dangling pointer when the object it points-to is deallocated, leaving the pointer pointing to dead memory. In this work, we focus on temporal errors rather than spatial ones. Thus, pointers that point outside their expected memory buffer (e.g., due to incorrect pointer arithmetic or a buffer overflow) are not considered to be dangling pointers, even if they end up pointing to dead memory.

Store address. A dangling pointer is always stored in *live* (allocated) memory or in a register. It may be stored anywhere in live memory: in live heap, in the current stack, in memory-mapped files (e.g., loaded using `mmap`), or in the data segment of any of the modules loaded in the program's address space. If the memory range containing its store address is deallocated (i.e., freed, unmapped, popped out of the stack), the dangling pointer is destroyed.

Points-to address. By definition, a dangling pointer points-to deallocated memory at creation. Later in its lifetime, it may point to a different object that resuses the memory vacated by the object the dangling pointer was meant to point-to. Although, strictly speaking, at this time the pointer is no longer "dangling", we consider it a dangling pointer until it is destroyed. In addition, any pointer derived from a dangling pointer, e.g., by copying the dangling pointer or using pointer arithmetic on the dangling pointer, is also a dangling pointer. Dangling pointers can point to any memory region that can be deallocated, (i.e., heap, stack, mapped files). The large majority of use-after-free and all double-free vulnerabilities in the CVE database are caused by dangling pointers created by freeing heap memory. However, we found (and include in our evaluation) a use-after-free vulnerability in Apache where the dangling pointers are created by unmapping a library from the program's address space. Although possible, we have not found any vulnerabilities in the CVE database where a dangling pointer pointed to dead stack.

Dangling pointer bugs. We call a deallocation that creates at least one unsafe dangling pointer a *dangling pointer bug*. We differentiate between two classes of dangling pointer bugs, depicted in Figure 3. In the *non-sharing* bug (left subfigure) the same object contains the pointer used to deallocate the object (freer) and the unsafe dangling pointer (user). In the *sharing* bug (right subfigure) these two pointers are stored in different objects. A key difference is that a sharing bug indicates some lack of coordination: the object left with an unsafe dangling pointer does not know that the deallocation happened. Since coordination may be required from object types that may be spread across multiple method invocations and program modules, sharing bugs are typically harder to identify and fix than non-sharing ones. Among the real-world vulnerabilities we analyze in Section 4.1, we observed sharing and non-sharing bugs to be equally likely.

2.4 Architecture Overview

Early detection works on an execution of a program. It can be run online in parallel with the program execution or offline on a trace of the execution. Our Undangle tool works offline.

Figure 4 shows the architectural overview of Undangle. First, the program is executed on the given input inside a previously available *execution monitor*. The execution monitor is a plugin for the TEMU dynamic analysis platform [2], which is implemented on top of the QEMU open-source whole-system emulator [3]. The execution monitor runs any PE/ELF program binary on an unmodified guest operating system (x86 Windows or Linux) inside another host operating system (Linux on x86).

The execution monitor tracks the program execution and produces an execution trace, containing all executed instructions and the contents of each instruction's operands. In addition, it produces an allocation log with information about the allocation/deallocation operations (heap and memory-mapped files) invoked by the program during the run. For this, the execution monitor supports close to a hundred standard memory functions. We assume that the programmer provides information on any custom memory functions used by the program.

The execution trace and the allocation log are inputs to the early detection technique, which comprises three modules. The core of early detection is the *dangling detection* module, which detects the creation of dangling pointers, tracks their propagation until their safety window expires, monitors for dangling pointer uses, and outputs the information about the dangling pointer creation and the detection/use. To track the dangling pointers it leverages the *pointer tracking* module, whose goal is to output at any point during an execution where pointers are stored as well as their values. In turn, the pointer tracking module leverages a generic *taint tracking* module extended with a reverse map to identify which program locations have been derived from the same taint label. All three modules have been developed for this work. In addition, early detection leverages a previously available callstack tracking module [21].

We have also developed a *symbol reader*, which takes as input the program symbols (if available) and merges the symbol information into the output of the dangling detection module. Symbol information significantly improves the diagnosis of dangling pointer bugs by providing detailed callstack information, mappings from binary code to lines in the source code, and type information.

Implementation. We have implemented early detection in approximately 4,000 lines of Objective Caml code. The symbol reader module comprises an additional 1,000 lines of Objective Caml code and 1,000 lines of scripts. The execution monitor and the callstack tracking module were previously available.

3 Early Detection

This section details our early detection technique, which tracks dangling pointers until they are used, become unsafe, or are destroyed. Our technique is implemented as a stack of 3 modules, which we describe bottom-up in this section. First we briefly introduce the taint tracker module in Section 3.1, then we describe the pointer tracking module in Section 3.2, and finally the dangling detection module in Section 3.3.

3.1 Taint Tracking

Taint tracking [27, 28, 45, 50] is a widely used technique and we assume the reader is familiar with the basic concept. However, our taint tracking module includes an important addition. Previous taint tracking techniques are based on a *forward map* from program locations (i.e., bytes or words in memory and registers) to the set of taint labels assigned to the location[1]. Our taint tracking module implements an additional *reverse map* that associates a taint label with the set of all program locations that have that taint label in their taint set. The reverse map is updated simultaneously with the forward map to maintain synchronization. It enables fast lookup of all program state derived from some taint label, avoiding a scan of the forward map in exchange for some small processing during propagation. In early detection, the reverse map is used to quickly identify all dangling pointers pointing to a memory buffer when it is deallocated.

Our taint tracking module has been designed so that it can be used to implement different flavors of taint tracking. The generality is achieved by making the taint tracking module a polymorphic virtual class. Each application can instantiate it with their own taint label type, their own `propagate` method (implementing the taint propagation rules), and their own taint sources and sinks. The taint tracking module simply provides methods to operate on the forward and reverse maps. In early detection all properties of the taint tracking module are set by the pointer tracking module.

[1] In early approaches, the taint label was one bit and only one label was kept per location, but currently this varies with the application.

Figure 5: Lattice of pointer types.

```
type taint_label = {
  type : Ptr | Dangling
  root-type: HeapRoot |
    StackRoot |
    [...] |
    Pseudo-root
  root-addr : int64;
  icounter : int64;
  offset : int;
}
```

Figure 6: Pointer taint label.

3.2 Pointer Tracking

The pointer tracking module tracks throughout the execution where the pointers are stored and their values. It monitors how new pointers are derived by copying an existing pointer and by computing a new pointer from an existing one using pointer arithmetic. It also identifies root pointers that are not derived from any other pointers.

Comparison with prior work. Our pointer tracking module can be seen as a specialized type inference module with the simple type lattice in Figure 5. The reason we could not use more general type inference modules such as Rewards [40] and TIE [39] was that we need to be able to identify the (dangling) pointers at any point in the execution, e.g., to check in each instruction if a dangling pointer is being used. This requires a forward technique that identifies pointers as the execution progresses. With TIE we would have to solve a constraint system at each instruction (too expensive), and with Rewards we could miss pointers that have not yet been dereferenced. Our pointer tracking module uses a forward pointer inference technique based on taint tracking.

Overview. The pointer tracking module uses the taint information to mark which program locations store pointers. In addition, it tracks the current pointer value in a separate *value map*. At each instruction the pointer tracker checks if new root pointers, not derived from other pointers, are introduced and whether the instruction creates new pointers by copying or using pointer arithmetic. When a root pointer is found, the locations where the pointer is stored are tainted using the label shown in Figure 6, where the `root-type` and `root-addr` fields represent the kind and value of the root pointer respectively, the `icounter` field describes the instruction counter in the execution trace where the root was introduced, and the `offset` field captures the offset of this byte in the pointer (e.g., 0 for the least significant byte of a pointer and 3 for the most significant byte of a 32-bit pointer). When new pointers are derived, the taint (pointer) information from the source operands is propagated to the destination operands[2]. Since the taint label stores a root pointer value, at any point the reverse map can be used to identify all pointers derived from a root pointer. To minimize memory use, locations that no longer hold pointers are removed from the forward map (i.e., untainted) rather than setting their type to top.

Root pointers. Each memory buffer has an associated set of root pointers (often only one) from which all pointers to the buffer are derived. For a heap buffer, its root pointer is stored in the return value of the heap allocation function. The location of the return value (often the EAX register) is stored in the allocation log produced by the execution monitor. For the parameters and local variables in a function's stack frame, the root pointer is the stack pointer (stored in the ESP register) at the function's entry point. To identify function entry points we use a callstack tracking module developed in prior work [21]. For statically allocated buffers, root pointers are global pointers, which we identify using debugging symbols and relocation tables, if available[3].

To identify global pointers when no symbol information or relocation tables are available, at each instruction the pointer tracking module identifies pointers being dereferenced. If a pointer being dereferenced is not yet tainted, it means that no root pointer is known for it, so the pointer tracker taints it setting the root type to pseudo-root and the root address to its current value.

Value tracking. The taint information marks which program locations store pointers, but it does not capture the current pointer value, which the tracker may need to output at any execution point. In an online setting, the pointer value can be simply obtained by reading the memory that stores the pointer. However, when operating on execution traces this value needs to be tracked and two challenges need to be addressed: our traces only contain the value of the instruction's operands before the instruction is executed and they only contain user-level instructions. To address the first issue, the pointer tracking module emulates the small set of x86 instructions used to derive new pointers (Table 1) to obtain the value of the destination operands after the instruction has executed. This emulation also identifies underflow and overflow in addition and subtraction, needed by some propagation rules (explained next).

[2]Instructions like `xchg` (exchange) have multiple destination operands

[3]PE executables (EXE files) running on a 32-bit Windows OS do not need relocation tables. They are the first module loaded into the virtual address space and are loaded at their preferred address.

Table 1: The x86 instructions that propagate pointers, their abstraction, and the associated propagation rules.

Instructions	Abstraction	Rules
mov, movs, push, pop	dst $\leftarrow$ src	move
xchg	t $\leftarrow$ src; src $\leftarrow$ dst; dst $\leftarrow$ t	exchange
add	dst $\leftarrow$ $\text{src}_1 + \text{src}_2$	add, default
inc	srcdst $\leftarrow$ srcdst $+ 1$	nop, default
sub	dst $\leftarrow$ $\text{src}_1 - \text{src}_2$	sub, default
dec	srcdst $\leftarrow$ srcdst $- 1$	nop, default
lea	dst $\leftarrow$ (disp + index) + base	add, default
nop*[4]	-	nop
All other	dst $\leftarrow \top$	default

The lack of kernel instructions means that a register or memory range, that when passed to the kernel contains a pointer, may no longer contain a pointer when the kernel returns execution to user-level. To address the second issue, when a user-level instruction reads a tainted operand, the tracker compares the value of the operand in the trace with the tracked pointer value. If they are not equal, the operand's taint is cleared to indicate that the value has changed unexpectedly and it is unlikely that it still holds a pointer. This is a conservative approach because the value returned by the kernel could be different but still a pointer. In those cases, if the returned pointer is dereferenced later in the execution the tracker will mark it as a pointer again.

Pointer propagation rules. At each instruction in the execution, pointer propagation rules are applied to identify new pointers being created by copying existing pointers or as a result of pointer arithmetic. Table 1 describes the mapping between x86 instructions that could produce a new pointer and the specific pointer propagation rules that the instruction may trigger. There are five classes of rules. The `move` rule which copies (parts of) a pointer; the `exchange` rule; the pointer arithmetic rules `add`, `sub`; the `nop` rule, which leaves the taint information as it is; and the `default` rule, which removes any pointers in the operands written by the instruction.

A fundamental difference between the `move` rule and the arithmetic rules is that pointer arithmetic has to be performed using the complete pointer, while a program can move or copy pointers completely or in chunks. For example, a program could copy a 32-bit pointer byte-a-byte using four instructions that write to consecutive memory locations and the 4 destination bytes would still represent a pointer. Such unaligned copies happen often in functions that copy memory, e.g., `memcpy`. Thus, the `move` propagation rule operates on program locations (i.e., bytes), while the arithmetic propagation rules operate on instruction operands.

Arithmetic instructions need two propagation rules to differentiate between valid and invalid pointer arithmetic. We consider only two valid pointer arithmetic operations that return a new pointer: adding an offset to a pointer without overflow, and subtracting an offset from a pointer without underflow. All other arithmetic operations on a pointer (e.g., adding two pointers or subtracting two pointers) are invalid and do not return a pointer. Invalid pointer arithmetic operations trigger the default rule.

3.3 Dangling Detection

The dangling detection module is responsible for identifying the creation of dangling pointers, detecting dangling pointers when the safety window elapses, detecting any use of a dangling pointer that may happen before the safety window elapses, and outputting diagnosis information for the detected dangling pointers.

Dangling pointers may be created and destroyed every time memory is deallocated. The dangling pointer module uses the allocation log, output by the execution monitor, to identify when heap memory is freed and files are unmapped, and the callstack tracker to identify when a stack deallocation happens (i.e., the stack pointer is incremented). For each deallocation, the dangling detection module first destroys any pointers stored in the deallocated memory. Then, for each heap deallocation and file unmapping it detects dangling pointers to the deallocated buffer. For efficiency, it does not detect dangling pointers to deallocated stack by default, since we did not find such vulnerabilities in the CVE database.

Dangling pointers are identified by querying the reverse map to obtain the list of all pointers in the program's state derived from the root pointers for the buffer. If the list is not empty it sets the type of those pointers to dangling. Then, it sets a detection callback for the current timestamp plus the size of the safety window, and stores the list of dangling pointers created, the buffer information, and the current callstack. For vulnerability analysis, the safety window size is set to infinite, and for testing it defaults to 5,000 instructions. We evaluate safety window size selection for testing in Section 4.3.

The pointer tracking module tracks dangling pointers between creation and detection the same way as non-dangling pointers. At each instruction, the dangling detection module checks if any detection callback has expired. If so, it uses the reverse map to obtain the list of dangling pointers that are still alive. Most dangling point-

[4]nop* represents instructions of different lengths that a compiler can use as no-operation instructions, as well as some instructions whose only side-effect is setting the `eflags` register, such as `or %eax,%eax`.

ers are short-lived and will be destroyed by the time the safety window expires. If any dangling pointer is still alive they are flagged and diagnosis information is output. Note that new dangling pointers introduced between creation and detection are also detected because the pointer tracking module considers any pointer derived from a dangling pointer to be also a dangling pointer.

Dangling pointers in memory metadata. The metadata used by memory management functions may include lookaside data structures that store pointers to deallocated buffers for fast reuse. If the lookaside structures are stored in live memory then the pointers they store may be flagged as dangling pointers. To avoid this, the dangling detection module deactivates pointer propagation inside memory management functions. We assume the allocation log contains all memory allocation/deallocation invocations by the program including custom allocators.

Dangling pointer uses. It could happen that the safety window size is set too large and a dangling pointer is used before detection. To avoid missing those dangling pointers, the dangling detection module checks if the current instruction dereferences or double-frees a dangling pointer. For this, it queries the pointer tracking module to determine if any memory addressing operand used by the instruction stores a dangling pointer. If the instruction is the entry point of a deallocation function, it also checks if the address parameter is a dangling pointer. Any dangling pointer used is flagged.

Contextual information. For each dangling pointer used or flagged as unsafe, the following diagnosis information is output to assist in the analysis: (1) whether it is a use or an expiration of the safety window; (2) if a use, whether it is a use-after-free or double-free vulnerability; (3) a description of the deallocated buffer (e.g., address, size, callsite, deallocation timestamp); (4) the identifier for the thread that created the dangling pointers (i.e., invoked the deallocation) and if used, for the thread that used the dangling pointer; (5) the callstack at creation time and if used, at use time; (6) the list of dangling pointers created by this deallocation; (7) the list of dangling pointers created by this deallocation and still alive at detection/use; and (8) the memory region containing each dangling pointer in the previous two items (register, heap buffer, stack frame, module). If program symbols are available, they are used to enhance the callstack and to obtain type information for the heap objects. For the latter, the object allocation/deallocation site is mapped to a source file and line number and the allocation type at that line in the source code is obtained.

Table 2: Vulnerabilities used in the evaluation.

Name	Program	Vuln. CVE	Type
apache	Apache 2.2.14 mod_isapi	2010-0425	uaf-m
aurora	Internet Explorer 7.0.5	2010-0249	uaf-h
firefox1	Firefox 3.6.16	2011-0065	uaf-h
firefox2	Firefox 3.6.16	2011-0070	dfree
firefox3	Firefox 3.5.1	2011-0073	uaf-h
ie8	Internet Explorer 8.0.6	2011-1260	uaf-h
safari	Safari 4.0.5	2010-1939	uaf-h
ie7-uf	Internet Explorer 7.0.5	2010-3962	uf

4 Evaluation

In this section we evaluate our early detection approach. The evaluation comprises two parts. First, we evaluate Undangle for vulnerability analysis. We apply Undangle to diagnose 8 vulnerabilities (Section 4.1) and detail two of those vulnerabilities in a case study (Section 4.2). The vulnerability diagnosis results show that Undangle produces no false negatives and that it enables us to understand the common root cause of two use-after-free vulnerabilities in Firefox, which was missed by the reporter of the vulnerabilities and the programmers that fixed them, leaving the program vulnerable to variants of the reported attacks.

Then, we evaluate Undangle for testing on two recent versions of Firefox (Section 4.3). Undangle flags 6 unique dangling pointer bugs: one which we believe is unsafe, two in Windows libraries where we cannot determine if they are unsafe or latent, and three that we believe are only latent. The results show that the false positive rate is low and that the output diagnosis information makes it easy to understand and fix the bugs.

Vulnerabilities. Table 2 shows the 8 vulnerabilities we use to evaluate Undangle for vulnerability analysis. These 8 vulnerabilities were selected because all of them have an exploit available in public databases[5] [4]. One of them, the *aurora* vulnerability, was exploited in a high profile attack on Google's and Adobe's corporate network [52]. Seven of the vulnerabilities are in popular web browsers. The remaining one is in the `mod_isapi` module of the Apache web server. All vulnerable programs are run inside a Windows XP Service Pack 3 guest OS.

The last column in Table 2 shows the vulnerability type. Six of them are use-after-free vulnerabilities and one is a double free. The final one is an underflow that was incorrectly flagged as a use-after-free; we use it to demonstrate how Undangle can verify doubtful disclosures. In 5 of the 6 use-after-free vulnerabilities, the dangling pointers

[5] Vulnerabilities privately reported through bounty programs [34] are assigned a CVE identifier but typically have no public exploit.