CENSUS
IT Security Works

Project Heapbleed

Thoughts on heap exploitation abstraction (WIP)

CONFERENCE
ZERONIGHTS 2014
NOVEMBER 13-14

PATROKLOS ARGYROUDIS
CENSUS S.A. argp@census-labs.com **www.census-labs.com**

Who am I

- Researcher at CENSUS S.A.
    - Vulnerability research, reverse engineering, exploit development, binary and source code auditing, tooling for these
- Before that I was working (postdoc) on applied cryptography at Trinity College Dublin
    - Designing, implementing, attacking network security protocols
- Heap exploitation abstraction obsession; joint work with huku (who would be here if Greece didn't have compulsory military service ;)

Warning

- No pictures
- No diagrams
- No charts
- (almost) No math
- Lots of text (I promise to try not to just read slides)
- Perpetual work-in-progress

Outline

- Introduction and motivation
- Related work
- Types and categories of heap bugs
- Heap attacks and exploitation abstraction
    - Identifying and defining reusable primitives
- Heap exploitation modeling

Motivation

- Heap bugs are the most common type of bugs
- Understanding of
  - allocator’s medata,
  - allocator’s allocation/deallocation algorithms,
  - how the target application interfaces to the allocator,
  - how application-specific data are placed on the heap,

  ⇨ in order to create conditions aiding exploitation
- Complicated bugs
- Increasingly sophisticated mitigation technologies

Objective

- Heap exploitation is becoming increasingly harder and more complicated
- Need to find ways to reduce the time required for heap attacks/exploitation
- Our goal is not to perform an academic exercise, i.e. create a formal model and publish
- Practical, reusable heap attack primitives that reduce exploit development time/effort

Abstraction

- Abstraction and the definition of reusable primitives is a valuable tool to tackle complexity
- “Design patterns” in software engineering
  - Reusable solution to a commonly occurring problem within a given context
- Sure, (heap) exploitation is much more complicated than writing software (it is) but the concept applies
- Some previous work on exploitation* abstraction

* The term “exploitation” in this talk is used in the context of memory corruption vulnerabilities

Related work

- Exploitation blueprint (Valasek, Smith)
  - Examples on modern common applications (bug to exploit)
  - Showcased reusable techniques
- Automated exploitation grand challenge (Vanegue)
  - Goal: reduced or no human interaction
  - Identified categories of exploit primitives
  - Model heap operation with a probabilistic transition system (Markov chains)
  - Random walks to reach exploitable heap states

Related work

- Weird machines (Flake, Bratus, et al)
  - State machine of the target after memory corruption
  - New (unexpected by the developer) states now reachable
  - Violation of security specification, i.e. exploitation
- Modeling of exploitation (Miller)
  - Finite set of primitives for transitioning between the states of a target under a memory corruption bug
  - Exploitation techniques combine these primitives to reach desired end states

Heap bugs

- Buffer overflow
- Use-after-free
- Dangling/stale pointer
- Double free

Buffer overflow

- Allocating a buffer on the heap
  - Perhaps with a wrong size due to a wrong calculation
  - Then writing more data to it
- Writing to a heap array with a for loop
  - That relies on a wrongly calculated loop limit

```
int a, b;
if(a > 0)
    char *dest = (char *)malloc(a);
memcpy(dest, src, a - b);
```

Dangling/stale pointer

- Have an allocated heap item
  - For example, an object (instance of a class)
- Have a pointer to it
- Perform an action that frees the heap item
  - Out-of-sync reference count of the heap item
  - Without invalidating the pointer
- The pointer is now dangling/stale
  - Pointing to a free heap "slot"
- Somehow the slot is reclaimed with data/object of your choosing (must be of the same size as the freed one)

Use-after-free

- What follows from a dangling/stale pointer bug
- The “slot” is usually reclaimed via spraying
  - The bug may allow reclaiming without spraying
- Depending on what the pointer was pointing to and with what the heap “slot” is reclaimed
  - Object pointer
  - Vtable pointer
- Just dereferencing the pointer may not cause a crash (unless heap integrity tools are used)

Double free

- The deallocation API call (e.g. free()) is called twice on the same memory address
- Depending on the allocator may or may not lead to corruption of its metadata
  - Linked-list-based allocators
  - Bitmap-based allocators

```
char *dest = (char *)malloc(n);
if(some_condition)
    free(dest);
free(dest);
```

Attacking heap managers

- Interfacing to the allocator
- Heap arrangement / heap feng shui
- Metadata attacks
- Adjacent region attacks
- Application-specific data attacks

Interfacing to the allocator

- As the attacker we don’t have direct access to the allocator’s API
- We can only allocate/deallocate indirectly via the target application’s exposed functionality
  - Operating system kernel: system calls, IOCTLs, opening/closing devices, drivers’ APIs
  - Browser: Javascript, VBscript, ActionScript
  - Media player: Metadata tags, containers within containers

Enumerating interfaces

- We need to a way to trace allocations and frees while interacting with the target application
- Debugger/programmatic debugger
  - Breakpoints at allocator’s malloc-like and free-like functions
  - Logging details and continuing
    - Size of allocation
    - Returned address of allocation
    - Address to be freed
    - Backtrace
  - Quite slow and error prone for real targets

Dynamic interface mapping

- Utilize a dynamic binary instrumentation (DBI) framework like PIN or DynamoRIO
  - Many public examples available, everybody has their own
  - Image based filtering
  - Can be tweaked to be faster and less error prone than a debugger
  - Only for userland target applications
- Kernel module that hooks kernel’s malloc-like and free-like functions
  - A lot of noise
  - Manual stack unwinding to create filters
  - Current version not very polished, but works

Static interface mapping

- Very useful to have the sizes of objects/structures
  - To target reclaiming free “slots” on the heap
- Source code of target and/or debug information (e.g. PDB/DWARF files) are sometimes available
- We can parse the source code or the binary files with the debug data for the sizes of object/structures
- Clang for source code
- PDB/DWARF parsers for binaries with debug information
  - Microsoft’s DIA (Debug Interface Access)
  - lldb.utils.symbolication Python module

Static interface mapping

- How to reach the allocations of the identified interesting objects/structures?
- We can use basic binary/source static analysis to find possible call paths between the function that does the allocation and a function we can interface to (Javascript API, system call, etc)
  - Clang
  - IDA/IDAPython
  - Understand
- Fast and imprecise; no constraint collection/solving and/or symbolic/concolic execution (more on this later)

Interface primitives

- Primitive #1: Allocate
- Primitive #2: Free
- Primitive #3: Allocate controlled size
- Primitive #4: Allocate controlled type

Mitigation: ProtectedFree

- Microsoft has introduced a new heap exploitation mitigation in Internet Explorer that breaks primitive #2
- That is, our ability to interface from Internet Explorer to the underlying allocator’s free operation
- Per thread list that holds heap “slots” waiting to be freed
- A free operation adds to the list instead of actually deallocating memory (mark-and-sweep GC)
- Introduces non-determinism to the interface

Heap arrangement

- Depending on the bug, especially if it is a buffer overflow, we need to be able to arrange the heap in a favorable (to our goal) way
- When the bug is triggered the heap must be in a predictable state to position our data
- “Heap feng shui” (Sotirov) for web browsers
- Understand the allocator’s behavior
  - Runtime observation
  - Reversing it’s allocation/deallocation functions
  - E.g.: FIFO, the first heap item freed is the first returned

Heap predictability

- At any random given point in time the heap is in an unpredictable state for us
- Using the interface primitives and our understanding of the allocator's behavior we build primitives that help us bring the heap in a predictable state, e.g.
  - A number of same-sized/typed allocations to "defrag" the heap and get fresh heap "slot" containers (e.g. pages)
  - Subsequent ones contiguous
  - Free every other allocation to create free "slots"
  - Just an example, study your target allocator

Arrangement primitives

- Primitive #5: Force contiguous allocations
- Primitive #6: Create holes (free “slots”)
- Primitive #7: Reclaim a free “slot”

Mitigation: g_hIsolatedHeap

- Heap exploitation mitigation in Internet Explorer that breaks primitive #7
- Our ability to reclaim a “free” slot
- Different heap for certain objects deemed probable of being involved in use-after-free vulnerabilities
- The obvious bypass here is of course to find a suitable to our goal object that is allocated on the isolated heap
- As all mitigations, this should be viewed in tandem with the others (i.e. ProtectedFree)

Metadata attacks

- Building on heap arrangement primitives, we can position controlled allocations next to memory used by the allocator for its internal operation and bookkeeping
  - Since heap overflows are quite common
  - Or other ways, e.g. arbitrary inc/dec, etc
- Corrupted metadata force unexpected allocator behavior that can lead to exploitable conditions
- These are obviously highly specific to the target allocator
- However since most allocators follow similar designs, experience has shown that ideas behind attacks are reusable

Unlinking attacks

- Original unlink() attack by Solar Designer (2000)
  - Old glibc unlink attack
  - Windows kernel unlink attack

```
unlink(P, BK, FD)
{
  BK = P->bk;  // what
  FD = P->fd;  // where
  FD->bk = BK; // *(where) = what
  BK->fd = FD; // *(what) = where
}
```

```
Unlink(Entry)
{
  Flink = Entry->Flink; // what
  Blink = Entry->Blink; // where
  Blink->Flink = Flink; // *(where) = what
  Flink->Blink = Blink; // *(what) = where
}
```

Force-return used attack

- Some allocator designs are not linked list based
  - jemalloc is a widely used bitmap based allocator
- Arrays (bitmaps) are used to represent heap memory areas
  - Array elements are used to represent heap “slots”
  - E.g. value of 1 for free, 0 for used
- Metadata corruptions lead to controlled indexes
- Indexing is mainly used to find the first free “slot”
- We can force the allocator to return an already used “slot”

House of Force

- Phantasmal Phantasmagoria's Malloc Maleficarum, compendium of glibc heap exploitation techniques
- House of Force has some strict requirements, but is currently unpatched
  - Top chunk metadata (size) corruption (top chunk represents the heap as a whole and grows/shrinks in size)
  - Size-controlled allocation (influences the value of the returned heap item)
  - Another allocation (returns the heap item)
- We force the allocator to return an arbitrary address

Metadata attacks primitives

- Primitive #8: Unlink
- Primitive #9: Force-return used
- Primitive #10: Force-return arbitrary

Adjacent region attacks

- We build on the “force contiguous allocations” and the “allocate controlled size/type” primitives
- Goal: place a vulnerable allocation (buffer/object/structure) we can overflow from next to a victim allocation we will overflow onto
    - That will aid us in exploitation
    - E.g. string/array/vector object that we can corrupt its size field
    - E.g. (virtual) function pointers

Application-specific data attacks

- Heap exploitation mitigations are becoming increasingly sophisticated and effective
- Generic exploitation approaches relying on metadata corruption are either
  - Already patched/mitigated
  - Patched/mitigated as soon as they become public
- Our target application (that uses the allocator) has objects/structures with useful to exploitation data
  - Function pointers are the canonical example of course
- Replace “function pointer” with X

Function pointers, or X

- Where X is any useful (to exploitation) construct
- Develop heuristics to search for X during runtime in the heap mappings of the target
  - Function pointers are easy, others (e.g. vectors) quite possible too
- Use pageheap-like functionality to get the backtrace of the allocation of the construct
  - We know where it gets allocated
  - We can find a call path to there from an interface point
- Now we know how to allocate useful constructs

Application-specific :) primitives

- Primitive #11: Force adjacent region allocations
- Primitive #12: Allocate useful construct

Heap exploitation modeling

- The identified primitives form a methodology that can be manually applied when investigating a new target
- How can we automate this methodology as much as possible?
  - Read “automate” as “reduce human interaction”
- The first step is to model the heap allocator
  - What about the next allocator?
  - Do we need to categorize the allocators and model then?
  - Will the model(s) be practically useful?
- Describe the identified primitives in this model

Simple model

- Model the heap as an array
- Heap “slots” are array elements
- Heap reads are array accesses
- Heap writes are array updates
- Metadata? Allocated or free?
- Another array (bitmap) holding state
- No straightforward modeling of more complicated metadata (or their corruption)
  - Linked-lists and unlink attacks for example
  - Basically we need an array for every metadata variable/pointer

Deterministic finite automata

- A finite set of states (Q)
- A set of symbols (S, input events, aka alphabet)
- A set of transition functions (T)
  - t e T : Q x S -> Q
- A start state q e Q
- A set of final (or accepting) states F (subset of Q)
- “Stop” or “dead” states are the states that are not accepting, i.e. return themselves for any input

Example (from Wikipedia)

- DFA: binary input, input must contain even number of 0s
- Q = {s1 (even 0s), s2 (odd 0s)}
- S = {1, 0}
- q = s1
- F = {s1}
- t = {t1}

t1	0	1
s1	s2	s1
s2	s1	s2

DFA-based model

- The allocator’s metadata are modeled as the DFA’s transitions
- The user data placed on the heap (“slots”) are the input alphabet (symbols)
- Metadata corruptions
    - Corruption of the DFA’s transition tables
    - Different (than expected) output state for the same input state and input symbol
    - Attacker controls the state the DFA is in
- Data (application-specific, function pointers, etc) corruptions
    - Corruption of the input symbols
    - Attacker controls which transition function is applied, so therefore indirectly the state the DFA will reach

DFA-based model

- We can use proof by induction to show (prove) that a property we are interested in is true (holds)
    - For example that given an alphabet and a DFA that certain states are reachable
    - Which transitions must be corrupted and how
    - Induction: prove base step (case 0), hypothesis (case 0 to n), prove inductive step (case n+1)
- DFAs can be used for automated theorem proving
    - We can check invariants for the set of transitions

Practical considerations

- It’s not realistic to manually model all allocators we are interested in
- DBI PIN tool (Moloch) to automatically construct the deterministic finite automaton based on observed data, metadata, transitions
  - This however does not provide a fully representative model of the allocator
  - Manual fine tuning of the model based on our understanding of the allocator
  - Remember that the goal is not full automation, but “reduced human interaction”

?

QUESTIONS

Thank you!

CENSUS
IT Security Works