Letting your fuzzer know about target's internals

Rodrigo Rubira Branco (BSDaemon)
Senior Vulnerability Researcher
rodrigo *noSPAM* kernelhacking.com

Agenda

- Objectives / Introduction
- Fuzzers and misconceptions
- Into software flaws
- Target's internals
- Implementation details and limitations
- Future

EVERYTHING in 60 MINUTES!

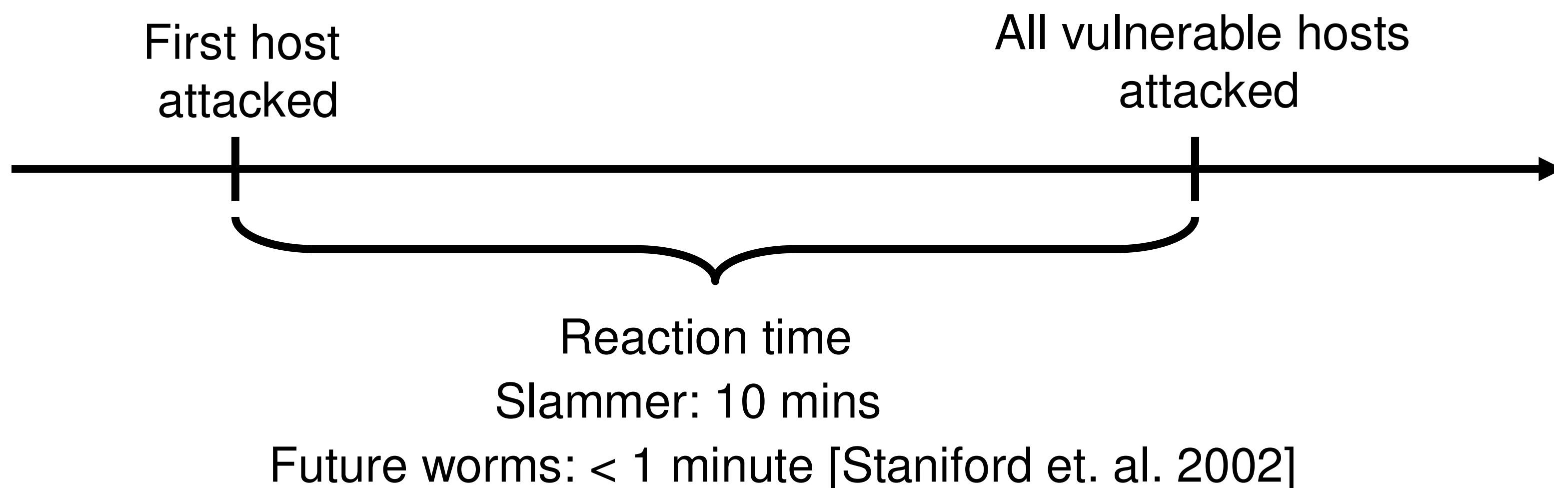
Objectives

- Show the added value of Hacking
- Demonstrate how fuzzers works, and why/when they are useful (or not)
- Explain what are the target useful internals information
- Explain how a debugger works
- Analyze security bugs and useful informations to feed back a fuzzer

Security nowadays

- Buggy programs deployed on critical servers
- Rapidly-evolving threats, attackers and tools (exploitation frameworks)
- Lack of developers training, resources and people to fix problems and create safe code
- **That's why we are here today, right?**

Security nowadays – 0day challenge



<u>"0day Statistics</u>
Average 0day lifetime:
348 days
Shortest life:
99 days
Longest life:
1080 (3 years)"

\- Justine Aitel

Introduction – What is a fuzzer?

- "**Fuzz testing**, **fuzzing**, **Robustness Testing** or **Negative Testing** is a software testing technique that provides random data ("fuzz") to the inputs of a program. If the program fails (for example, by crashing, or by failing built-in code assertions), the defects can be noted.

  The great advantage of fuzz testing is that the test design is extremely simple, and free of preconceptions about system behavior."

  - » Source: Wikipedia

Fuzzer - Misconceptions

- In the definition itself:

  "If the program fails (for example, by crashing, or by failing built-in code assertions), the defects can be noted." -> What if the program does not fail, but, for example, a memleak occur?

- In the code coverage principle:
  - It is important to fuzz other portions of the code (i.e. application expecting "auth: " in the beginning of the buffer)
  - It does not tell you how good are the fuzzer (90% of code coverage may spot just 10% of the bugs if they miss important security-related constructions)

- In the way it is done nowadays:
  - 'Dumb' fuzzers -> Really bad input streams
  - Code coverage are based in static analysis and function flows
  - Most of the fuzzer solutions, are missing the target's internal information

Fuzzing – Actual State

- Full fuzzer uses a protocol specific (think RFC) to the target program and works only for that protocol (i.e.: SMTP fuzzer)

- Mutation fuzzer (sometimes called capture/replay) starts with some known good data, changes it somehow, and than repeatedly delivers mutations of that data to the target.

Source: **Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing**

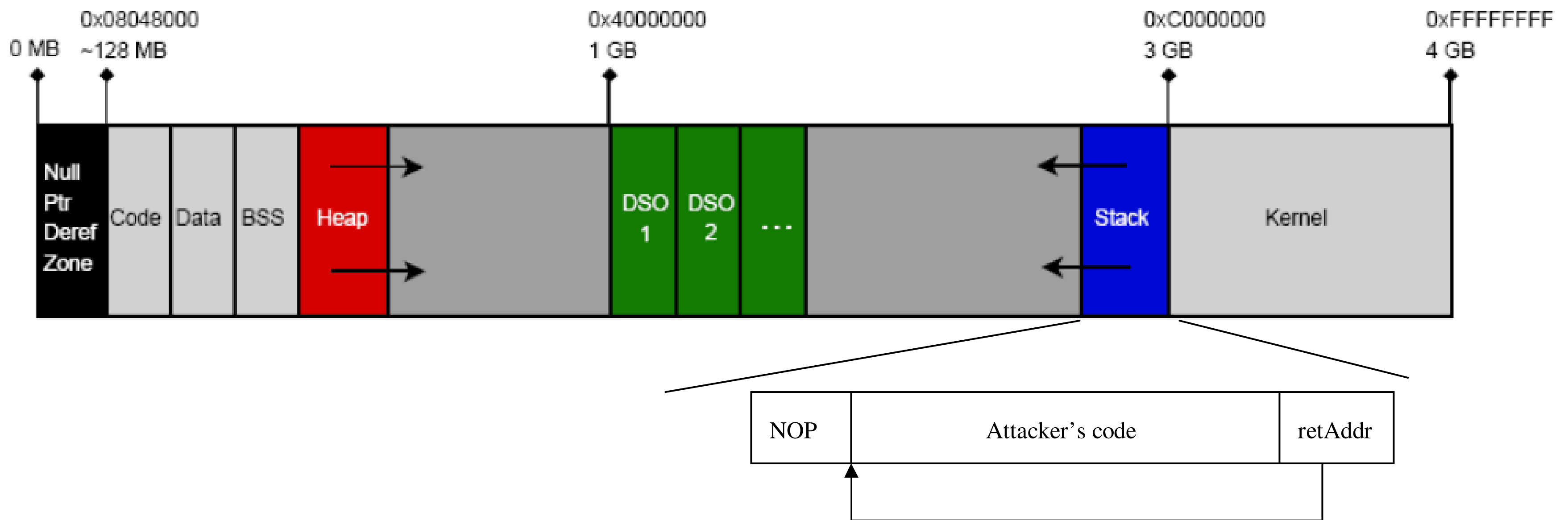
Evolutionary Fuzzing

- Use static analysis to feedback the fuzzer and achieve code coverage
- The fuzzer is responsible for changing the input data based on the information returned by the fuzzer, thus, learning the underline protocol
- Use sessions based on the protocol graph and coverage – tieing together multiple requests

Into software flaws – Low level vulnerabilities

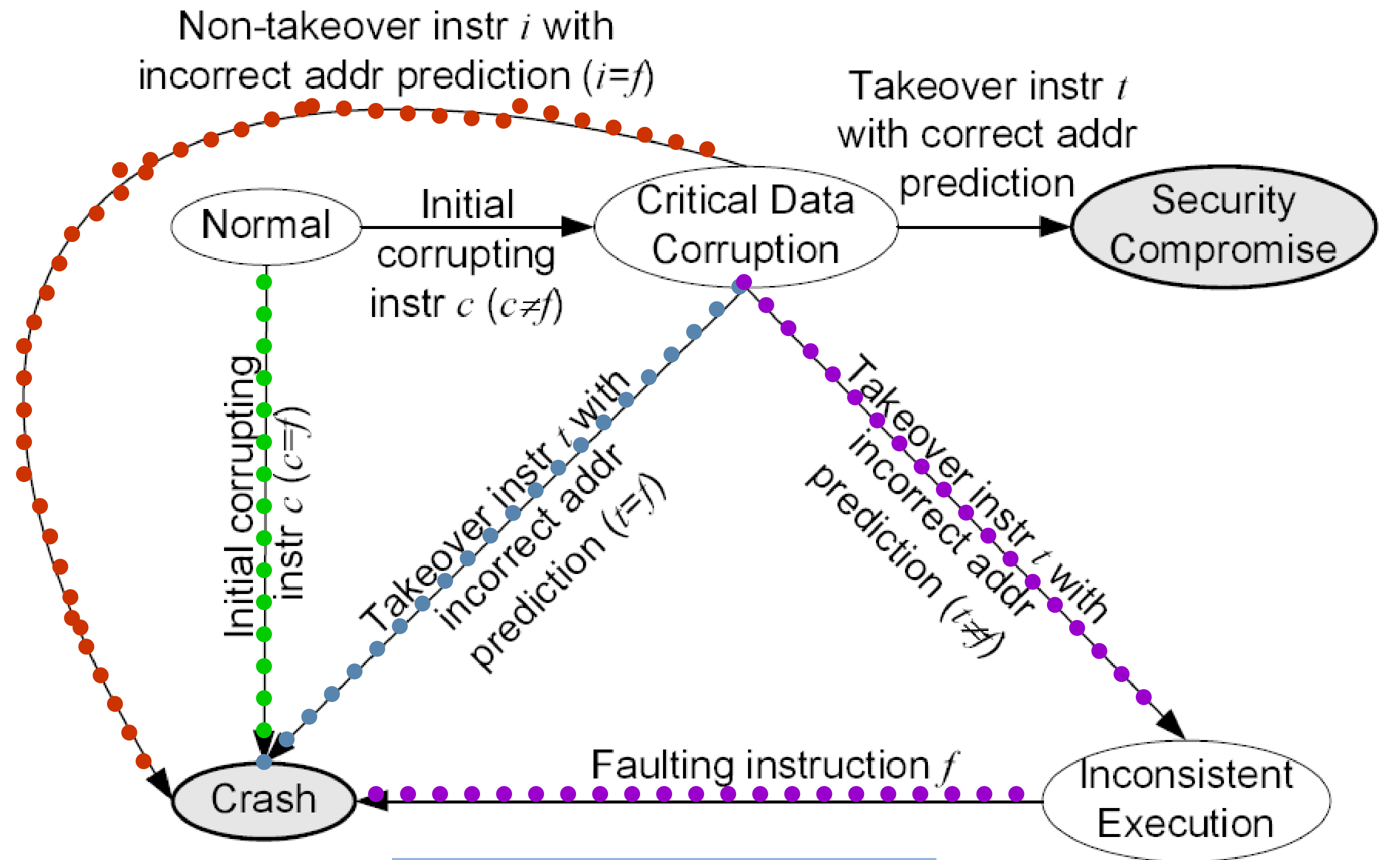
- Memory Corruption Vulnerabilities
  - Popular means to take control of target program
  - 49% of all attacks in 2006 (yeah, we all know that web-based vulnerabilities are evolving)
  - Successful attacks cause a remote code execution
  - Attack techniques: stack and heap overflows, integer problems (leading to overflows or memory disclosure) and others
- Trigger the vulnerability will lead to program crash
  - Fuzzers usually detect a flawed application when it crashes
  - As said, they miss many other cases (memleaks and syncing problems)
  - ASLR-based systems turn this even more unpredictable in the real-world

Memory corruption



State Transition for Memory Corruption

- Case 1 (green): Format String
- Case 2 and 3 (red and blue): buffer overflow
- Case 4 (purple): unpredictable



Source:
Automatic Diagnosis and Response to Memory Corruption Vulnerabilities

c: corrupting instruction
t: takeover instruction
f: faulting instruction

So, what?

- Legitimate assumption:
  - To change the execution of a program illegitimately we need to have a value being derived from the attacker's input (which we call: controlled by the attacker)

- String sizes and format strings should usually be supplied by the code itself, not from external, un-trusted inputs.

- Any data originated from or arithmetically derived from un-trusted source must be inspected.

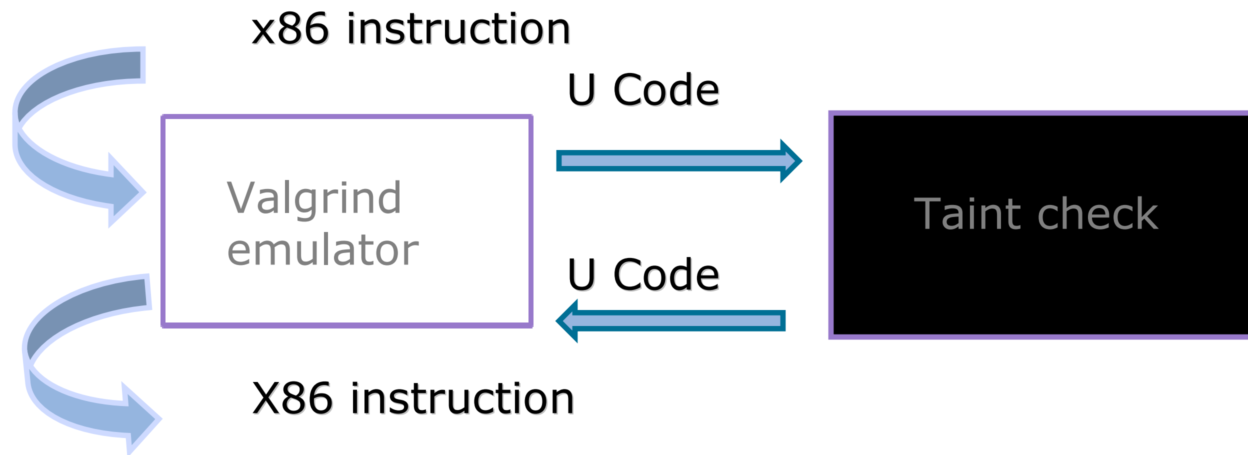
Taint Analysis

- Tainted data: Data from un-trusted source
- Keep track of tainted data (from un-trusted source)
- Monitors program execution to track how tainted attribute propagates
- Detect when tainted data is used in sensitive way
- Taint check perform dynamic taint analysis on a program by running a program in its own emulation environment.

Real-world applications

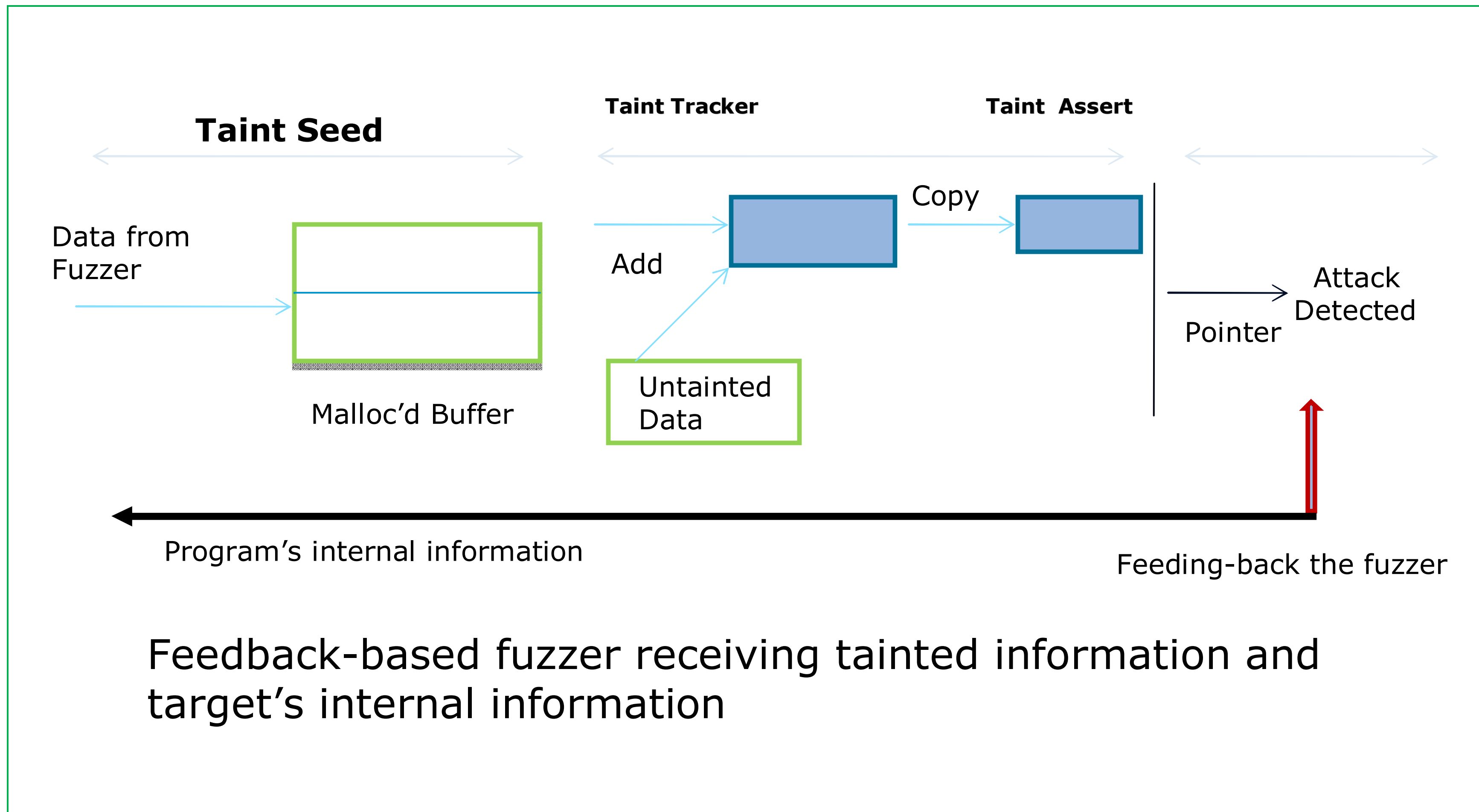
- Jin Chow –"Understanding data lifetime via whole system emulation" –presented at Usenix"04.
- Created a modified Bochs (TaintBochs) emulator to taint sensitive data.
- Keep track of the lifetime of sensitive data (passwords, pin numbers, credit card numbers) stored in the virtual machine memory
- Tracks data even in the kernel mode.
- Concluded that most applications doesn't have any measure to minimize the lifetime of the sensitive data in the memory.
    - Tks to Edgar Barbosa for the information

Taint Analysis



Source: Dynamic Taint Analysis for Automatic Detection Analysis

Feeding-back our fuzzer



Modified from: Dynamic Taint Analysis for Automatic Detection Analysis

Inheritance problems

Problem: state explosion for binary operations !

Application	Propagation Tracking	Inheritance Tracking
`mov %eax ← A` `mov B ← %eax`	`taint(%eax) = taint(A)` `taint(B) = taint(%eax)`	%eax inherits from A B inherits from %eax
`add %ebx ← D`	`taint(%ebx) |= taint(D)`	insert D into %ebx's inherit-from list

Events

Rare
e.g., malloc/free, system calls

Frequent
e.g., memory access,
data movement

Dynamic analysis

- Tracks program state
- Monitors memory writes and checks for violation of security condition
  - A given memory region may correspond to different program variables depending on program state
  - We need to keep track of memory mapping
- Tracks tainted data and its propagation
  - Consider the actual register values to determine the tainting of information:
    - If A is tainted and you ***AND*** it with B which is not tainted and B is 0 that means the result is not tainted anymore, since you cannot control the result of the AND. If B is 1 in this case, the result is also tainted
- Instrument binary in runtime (mutating the target)

Data Structures – Memory Tracking

- Used memory
  - Memory corresponding to program variables
- Control memory
  - Saved registers, return addresses, metadata encapsulating dynamically allocated memory regions (heap information blocks)
- Program State (function calls/return)
  - Local variable addresses are calculated and added to Used memory
  - Location of return address and saved registers are added to the control memory list
- Heap memory
  - malloc/free calls are hooked
  - Allocated memory is added to used memory
  - The heap metadata are added to the control memory

Feeding back the fuzzer

- When a instruction writes to a memory address, check:
  - If the address is in the used list
    - » Determines the variable it belongs to
    - » Checks if the value written comes from un-trusted source
    - » Validate if the allocation routine have been in somewhat controlled by the input (resource-starvation attacks)
    - » Validate the pair, malloc/free to spot memleaks
  - If the address is in the control list
    - » It is a memory corruption
    - » Feed-back the fuzzer with the program state and input value responsible for the overwrite

Run-time static analysis

- Avoid the needle of a single-step to track memory accesses
- Performance gain (we are fuzzing after all!)
- Remember, every {watch|break|point hit has a penalti:
  - Debuggee triggers the {watch|break}point – Trap to OS
  - OS gives the control to the Debugger
  - Debugger read the status of debuggee from the OS
  - Debugee context activated for the information gathering
  - OS reads the information
  - Debugger activated again, to receive the information

Heap View

- Gives information regarding the heap structures
  - Independent of the OS structures (we feed the fuzzer with the heap structure – actually supported: Linux – CORE Security released something for Windows)
  - Many heavy programs have its own heap allocators
- Visual view of the heap state
  - Important to detect memleaks
  - Useful to detect small overwrites that not crash the target (usually it will crash along the time, turning very difficult to determine what triggered that condition – c != f)
  - State of the bins
  - Very useful for later exploit construction

c: corrupting instruction
f: faulting instruction

Mutating the target

- It occurs altering the path of execution
  - Bypassing protection directives in runtime
  - Feedback the fuzzer with internal information regarding the protection (useful to find integer overflows)

- Can be done in kernel mode to instrument the OS kernel
  - Patching the original function
  - Finding inlined functions using static analysis

Genetic Programming

- Machine-learning approach to automatically creating computer programs by means of evolution
  - In the EFS fuzzing, it means create inputs based on the target internals
  - In my approach, it means modifying the target in such a way that the input fits the conditions, and if a vulnerability is found, feeding back the fuzzer with the needed information to create the input
- We need to model the call dependencies (program flow graphs) in order to be sure about the exploitability vectors
  - It is easy since we taint the input
  - Do the reverse lookup to find all the trigering vectors

Another problem solved...

- Conditions that does not exist in the default configuration or in the target configuration
  - I.E.: The vulnerability does exist just when the option ‘x’ is defined in the conf. File
  - Since when the debugger change the condition to ‘true’, it will spot the vulnerability, but since it’s not trigerable changing the input (the condition is not controlled by a tainted data) feedback the fuzzer with that information (i.e.: vulnerability detected if condition ‘y’ not controlled by the input do exist)
  - This question appeared when auditing a complex software, with many optional configurations

- Target mutation is needed when testing something not directly controlled by the attacker input
  - This appeared when auditing pop3 caching files
  - The email file is created by a SMTP server
  - The file is then readed by the pop3 server and then written to the cache

Why valgrind?

- “The Valgrind tool suite provides a number of debugging and profiling tools”
- Supports extensions, the plugin tools
- Able to instrument a program in runtime
  - Uses an intermediate language, VEX, which are a RISC-like language
- There is also a standalone version, created to support more architectures/OSes
  - Ported to be used on Windows

The need for intermediate languages...

- Assembly instructions have explicit operands, which are easy to deal with, and sometimes implicit operands:
  - Instruction: push eax
  - Explicit operand: eax
  - What it really does?
    - » ESP = ESP – 4 (a substraction)
    - » SS:[ESP] = EAX (a move)
    - » Here we have ESP and SS as implicit operands

      - Tks to Edgar Barbose for this great example!

Valgrind's Plugins

- pre_clo_init() -> VG_DETERMINE_INTERFACE_VERSION(my_pre_clo_init)
  - All the initialization will be done here
    - VG_(basic_tool_funcs) (my_post_clo_init, my_instrument, my_fini);
  - Set our handlers:
    - VG_(needs_malloc_replacement)()
    - VG_(track_new_mem_heap)()
    - VG_(track_new_mem_brk)()
    - VG_(track_new_mem_mmap)()
    - VG_(track_copy_mem_remap)()
    - VG_(track_change_mem_mprotect)()
    - VG_(track_pre_mem_read)()
    - VG_(track_pre_mem_write)()
    - VG_(track_post_mem_write)()
    - VG_(needs_syscall_wrapper)()
- post_clo_init() -> my_post_clo_init()
- instrument() -> my_instrument()
- fini() -> my_fini()
  - It will be called in the end of the process
  - Will provide a summary for our fuzzer

Instrument() function

```
static
IRSB* my_instrument ( VgCallbackClosure* closure,
                      IRSB* sbIn,
                      VexGuestLayout* layout,
                      VexGuestExtents* vge,
                      IRType gWordTy, IRType hWordTy )
{
```

Library Preloading Limitations

- Library preloading could be used in some cases to track the heap (malloc/calloc/realloc/free function hooks)

- It's very limited since will miss inlined functions, direct brk()/mmap() calls and static applications
  - Was used in the beginning of the implementation since it's easier to debug

- It changes the loading address of the libraries, thus, breaking the **heisenberg** principle

Distributing the Problem

- An address to the debugger is:

  Native_Addr.Node_Addr.Offset_from_beginning (each instruction counted by 1 don't matter it's size - we modify instructions)

- Because of that we can use a multi-threaded fuzzer to generate input data for ‘n’ targets (nodes in the cluster)
  - Saving the state, synchronizing or whatever to distribute also the generation work, but it’s not really needed unless there is a need for resume – because most of the ‘cpu-intensive’ work are been done in the target itself.

Algorithms

- Metaheuristic search algorithms are helping in the target analysis
- SAT solvers and other decision algorithms
- Address range inspection defined by the fuzzer, avoiding testing of not ‘needed’ areas of the program and receiving human-decisions

Case Study: Solaris Sadmin

- Solstice AdminSuite is a set of applications for distributed system administration. *sadmind* is a daemon used by Solstice Adminsuite to control the servers running Sun Solaris operating system.
- Vulnerability found, exploited and released by RISE Security in October/2008
- Two new vulnerabilities found by Secunia:
  - Secunia identifier SA32473, dated 2009-05-23
  - No details at ***ALL***

CVSS Scores

- Temporal score is **7.4 (remote heap overflow):**
  - Because the exploitability level of this vulnerability is **unproven (hummmm, not anymore…)**
- Temporal score is **6.9 (remote integer overflow):**
  - Because the exploitability level of this vulnerability is **unproven (hummmm, not anymore…)** and the complexity for exploitation **(really??)**.

What I hate in advisories?

- No details at all... They are used just as marketing stuff, not really to help the security community
- What I had? The previous vulnerability and exploit...

Results

- The heap overflow vulnerability:
  - Occurs in: __0fNNetmgtArglistNdeserialValueP6DXDRUiTCPc
  - The code:
    - » .text:0000F316 push eax
    - » .text:0000F317 push [ebp+arg_4]
    - » .text:0000F31A call _xdr_u_int **<- Tainted value (array size)**
    - » .text:0000F31F add esp, 8
    - » .text:0000F322 test eax, eax
    - » ...
    - » .text:0000F35E push dword ptr [ecx+408h] **<- Tainted value (array size will be used as parameter for the next call)**
    - » ...
    - » .text:0000F374 call ___0fNNetmgtArglistNdeserialValueP6DXDRUiTCPc

Results

- .text:0000C61D push dword ptr [ebp+arg_C] **<- Tainted value will be used as parameter for the next call (the allocation itself)**
- .text:0000C620 call _calloc **<- Buffer allocation**
- **...**
- .text:0000C687 call _xdr_bytes **<- The buffer is used for the xdr_bytes call, overwriting the array size with bytes_length from network**

Results2

- The integer overflow vulnerability:
  - Occurs in: __0fMNetmgtBufferFallocUiTB
  - The code:
    - .text:0000A306 cmp dword ptr [eax+4], 0 **<- If not allocated**
    - .text:0000A30A jz loc_A392 **<- Allocate**
    - **...**
    - .text:0000A328 mov ecx, [ebp+arg_0] **<- Reallocation**
    - .text:0000A32B mov eax, [ecx+8] <- **Current Size**
    - .text:0000A32E add eax, [ebp+arg_4] **<- Size from the XDR Header (taint it)**
    - .text:0000A331 mov esi, [ebp+arg_8] **<- block_size**
    - .text:0000A334 xor edx, edx
    - .text:0000A336 div esi <- **Size divided by block_size**
    - .text:0000A338 inc eax <- **+1**
    - .text:0000A339 imul esi, eax <- **Multiplying by block_size**
    - .text:0000A33C push esi <- **Overflowed integer will be allocated**

Related Projects

- **AddrCheck:**
  - » Monitor malloc/free, memory accesses
  - » Check if all memory accesses visit allocated memory regions

[Nethercote'04]

- **MemCheck:** AddrCheck + check uninitialized values
  - » Copying partially uninitialized structures is not an error
  - » Lazy error detection to avoid many false positives
  - » Track propagation of uninitialized values

[Nethercote & Seward '03 '07]

- **TaintCheck:** detect overwrite-based security exploits
  - » Tainted data: data from network or disk
  - » Track propagation of tainted data to detect violations

[Newsome & Song'05]

- **LockSet:** detect data races in parallel programs

[Savage et al.'97]

- **Memsherlock:** automated debugger

[Ning et al.'07]

- **N-Variant:** create variants of a system and examine it's behaviour

[Cox et al.'07]

Future

- I can't foresee the future!

- I'm in the early development stages, so many new challenges will be shown in the future
  - Combinational explosion
  - SAT testers limitations
  - Byte-level tainting -> Some false positives...

- The focus of this research are in the debuggers being attached to the target program to collect internal's information, not in the fuzzers itself (I did just simple implementations using the debuggers information regarding flow and internal structures)

End! Really !?

Rodrigo Rubira Branco (BSDaemon)
Senior Vulnerability Researcher
rodrigo *noSPAM* kernelhacking.com