Shuffler

Thursday, September 14, 2017

6:15 AM

**Shuffler: Fast and Deployable Continuous Code Re-Randomization**

Suggestion

* Shuffling partitions at a time to decrease the latency
* Shuffling a forwarding table rather than the actual memory space

Introduction

* Shuffler
* Shuffles every 50ms,
* 14.9% overhead on SPEC CPU
* Runs on user space along with target program
* Only needs minipal patch to the loader
* Operates by performing continuous code re-randomization at runtime, within the same address space as the programs it defends.
* Real-time deadline = shuffle period
* Requires a small global pause time of .3% as program threads continue to execute
* If the machine has a spare CPU core, shuffling at a faster rates will not be hindered
* Defends against:
  + ROP
  + Direct JIT-ROP
  + Indirect JIT-ROP
  + Blind ROP
* Main contributions:
  + Deplorability - runs without modifications to the source, compiler, linker, or kernel and minimal changes to the loader
  + Speed - introduce a real-time deadline. use asynchronous re-randomizataion architecture with low latency and overhead
  + Egalitarianism - avoids any expansion of the trusted computing base boostraping into a self-hosting environment
  + Augmented binary analysis - complete and precise analysis is possible leveraging info available from compilers

Background & Threat model

* Defenses against code reuse
  + Control Flow Integrity
    - Tries to ensure that every indirect branch taken by the program is in accordance with its control-flow graph
    - Coarse-grained CFI and fine-grained CFI can be bypassed through careful selection of gadgets
  + Code randomization
    - Performed at load-time to make the address of gadgets unpredictable
    - Module-level Address space layout randomization (ASLR) is currently deployed in all major os
    - Lead to a new form of attack called JIT-ROP in 2013.
      * Attacker starts with one known code address, recursively reads code pages at runtime with a memory disclosure vulnerability, then compiles an attack using gadgets in the exfiltrated code
    - No load time randomization scheme can stand against this attack

* Defenses against JITROP
  + At first it concentrated on preventing recursive gadget harvesting
  + Preventing direct disclosure of memory pages is insufficient
  + Indirect JIT-ROP shows that harvesting code pointers from data pages allows the location of gadgets to be inferred without ever being read
* Continuous re-randomization
  + can defend against JIT-ROP
  + Code needs to be re-randomized between the time it is leaked and when a gadget chain is invoked in order for the attack to fail since the gadget no longer exist,
  + Adding padding NOPs
  + Blind ROP targets servers that fork workers
    - Workers inherit the parent's address space layout
    - Brute forces the workers without worrying about causing crashes.

Threat Model

* Effective when defending long-lived processes and network-facing applications such as servers

Design

o 
shurne 
2 
o 
Libraries 
Figure I: Shuffler architecture. We use symbols and re- 
locations (O) for augmented binary analysis (I j, rewrite 
code into shufflable form (2), and asynchronously create 
new code copies at runtime (3), while self-hosting (4). 

* architecture
  + One load-time. Shuffler transforms the program's code using binary rewriting.
  + This is to track and update all code pointers at runtime
  + Transform all code pointers into unique identifiers and incices into a code pointer table
  + Asynchronously generating sandboxes
  + Shuffle period - a randomization deadline specified in milliseconds
  + Shuffler runs in a separate thread and prepares a new shuffled copy of code within this deadline (process is accelerated using Fenwick tree)
  + Vast majority of the process is async
    - Creating new copies of code
    - Fixing up instruction displacements
    - Updating pointers in the code table
  + Threads are briefly paused globally to atomically update return addresses (in the stack frame as well)
  + Binary rewrite to transform all code in a userspace application into a single code sandbox

Challenges

* Changing function pointer behavior
  + Failing to update even one pointer may cause the program to crash or fall victim to a use-after-free attack
  + Encrypts the addresses on the stack using XOR
* Augmented binary analysis
  + Analyzing program binaries that have additional information included by the compiler
  + Compiler preserve the symbol table
  + Linker preserve relocations
  + Require that
    - The compiler preserve the symbol table
    - The linker preserve relocations
  + Iterate though symbols and disassemble each one independently
* Bootstrapping into shuffled code
  + Shuffled code cannot start running until the code pointer table is initialized.
  + Shuffled and original code are incompatible if they use code pointers
  + Two stages:
    - Minimal stage - completely self-contained, and it can safely transform all other code
    - Shuffled second stage - erases the previous stage ( and all other original code)
      * Inherits all the data structures created in the first so that it can easily create new shuffled code copies.
  + Fully self-hosting.

Implementation

Figure 2: Overview of shuffled code at runtime, as Shuf• 
Her executes a shuffle pass. The old code is shown with 
solid lines and the new code with dotted lines. 

* Runs in userspace on x86-64 Linux
* Shuffles binaries, all the shared libraries that a binary depends on, and itself
* Shuffling process runs assynchronously in a thread without impeding the execution of the program's threads
* Supports many system level features, including shared libraries, multiple threads, forking, jmp, system call re-entry, and signals
* Exposes several debugging features: exporting shuffled symbol tables to GDB and printing shuffled stack traces

4.1 Transformations to Support shuffling

* Code pointer abstraction
  + Allocate the code pointer table at load-time and set the base address of the GS segment at it
  + Transform every function pointer at its initialization pint from an address value to an index into this table
  + Use relocations generated by the compiler and preserved by the linker flag -q to find all such code pointers
  + Every instruction which originally used a function pointer value is rewritten to instead indirect through the %gs table
  + Since x86 instructions can contain at most one memory reference, if there is already a memory dereference, use the caller-saved register %r11 as scratch space
  + For position-dependent jump tables, there is no register that can be safely overwritten. So use a thread-local variable allocated by shuffler as a scratch space
* Return-address encryption
  + Reuse the stack canary storage location for the XOR key
  + Add two instructions at the beginning of every function (to disguise the return address) and before every exit jump (to make it visible again).
  + After each call. Insert a mov instruction to erase the now-visible return address on the stack
  + Displacement reach
    - Normal call instruction has a 32-bit displacement and must be within +-2GB of its target to "reach" it.

Completeness of Disassembly

* There were few special cases issues
  + Dealing with inaccurate/missing metadata, especially in the symbol table
  + Handling special types of symbols and relocations
  + Discovering jump table entries and invocations
* Jump tables
  + GCC generates jump tables differently in position dependent and position-independent code (PIC)
  + Position-dependent jump tables use 8-byte direct pointers
  + PIC jump tables use 4-byte relative offsets added to the address of the beginning of the table
  + A heuristics that pairs function epilogues with function prologues decides which jump is used
* Bootstrapping and Requirements
  + Carefully bootstrap into shuffled code using two libraries (stage 1 and stage 2) so that the system never overwrites code pointers for the module that is currently executing
  + Defer to the system loader to create a valid process image, and then take over before the program, or even its constructors, begin executing
  + Stage 1 is called before any other via the linker mechanism -z initfirst
  + Makes sure all other constructors run in shuffled code
  + The last constructor to be called is stage 2's own constructor
    - Stage 2 creates a dedicated Shuffler thread, erase the original copy of all other code, and resumes execution at the shuffled ELF entry point
* Full shuffling Requirements
  + Compiler flags
    - Shuffler require the program binary and all dependent libraries to be compiled with -W1, -q, a linker flag that preserves relocations
    - Users must avoid -s, which strips symbols, and -fno-asynchronous-unwind-tables, which elides DWARF unwind information.
  + System modifications
    - The -z initfirst loader feature currently only supports one shared library, and libpthread already uses it.
    - To maintain compatibility with libthread, we patch the loader to support constructor prioritization in multiple libraries
    - Simplify shuffler's task and map all ELF PT\_LOAD sections into the lower 32 bits of the address space
    - Place the loader itself into the displacement region, preresolving its relocations with prelink
    - Disabled a manually constructed jump table in the vfprintf of glibc

Implementation Optimizations

* Bottleneck is in issuing many mprotect system calls
* Instead we maintain several buckets (64KB-1MB) and each function is place in a random bucket.
* Using Fenwick Tree, finding random memory space is optimized because you can specify the minimum size space you are looking for
* Stack unwinding is done by writing a custom library using DARF state machine to translate between shuffled and original address
* Binary rewriting - through careful block splitting, we can choose wheter incoming jumps execute or skip over new instructions as appropriate

.

Source instruction 
Transformation 
Lea funcptr 
call 
, 8) , *Ell 
cal Iq 
3mp r ax 
mpg 
*rbx, 
1 Q ndex, 
ax 
cal Iq • 
3mpq .*gs: ( ax' 
ifs • 
. oxS8 
mav ( *lax, E x 
mav *gs: 
xchg 
3mpq 
(a) Transforms to support the code pointer table 
Source instruction 
function begin —+ 
ret / 3mp 
call anything 
Transformatio n 
s: Ox28, *111 
xor 11, 
function begin 
mav s: Ox28, *111 
xor 11, 
/ 3mp ax 
call anything 
mav S Ox C, 
—S ( *rap) 
(b) Transforms to support return address encryption. 

Misstn s mbol sizes 
Fall-through symbols 
Overlapping symbols 
Symbol aliases 
Ambiguous n s 
Pointers to 
static functions 
function calls 
copy relocations 
'FUNC symbols 
Condi tional 
tail recurs ion 
Indirect tail rec. 
Findi um tables 
Description 
Internal GCC functions have a s mbol size of zero. 
Functions implicitly fall through to the following function. 
Some functions are a Strict subset Of an enclosing functtOtL 
Symbol tables have many names for the same function. 
One LOCAL multiple (b s in libm)_ 
For vwinters to functions within the same nuklule. the offset 
is known. and Object tiles contain no relevant relocations. 
GCC always generates a NOP after calls to r 
functions like imp. but omits unwind information. 
Object initialized in one library. then to another 
Return vxjinter to actual function to call (cached in PIX). 
not appear in normal GCC-generated code. Used in 
hand-c«led assembly by (low level Iock h). 
Difficult to tell apart from jump-table jumps. 
Jum tables are not Clearl delineated. 
How to handle 
Hard-code Sizes: 
start is 42 
Attach a copy of the following code 
Binary search for targets very carefully. 
Pick one representative 
Look up address resolved by the loader. 
Determine if lea instructions target a 
known symbol (not completely sound). 
Detect when at a NOP following a call 
and use unwind info from at the call. 
Track data not just code 
Statically evaluate from refs. 
can do XOR'ing both before and after. 
works whether or not the jump is taken. 
Use a function epilogLk heuristic. 
See the text for a discussion on this. 
Figure 4: Special cases in augmented binary disassembly. 