Georg-August-University

Seminar Report

Q: Exploit Hardening Made Easy

Author:
Seshagiri Prabhu

Supervisor:
Fabian Yamaguchi

A report submitted in fulfilment of the requirements for the seminar course: Intrusion and Malware Detection

in the

Computer Security
Institute of Computer Science

February 2015

Contents

Chapter 1

Introduction

Current practical implementations make compatibility and performance tradeoffs, and as a result it is possible to automatically harden existing exploits to bypass these defenses. This research work demonstrates the defenses as currently deployed can be bypassed with new techniques for automatically creating *ROP* payloads from small amounts of nonrandomized code. The research paper proposes a semantic program verification techniques for identifying the functionality of gadgets, and design a *ROP* compiler that is resistant to missing gadget types. In the research work, the authors have build Q, an end-to-end system that automatically generates *ROP* payloads for a given binary.

1.1 Writable $\oplus$ eXecutable

1.1.1 Advantages

Attackers have traditionally included shellcode (executable machine code) in their exploits as payloads. Since shellcode must be written to memory at runtime, it cannot be executed because of the $W \oplus X$ property. The main limitation of $W \oplus X$ is that it only prevents an attacker from utilizing new payload code

1.1.2 Limitations

The attacker can still reuse existing code in memory. Eg: *return-to-libc* attack.

1.1.3 Implementation

Hardware level using NX bit could be enabled through PAE addressing mode.

1.2 ASLR

1.2.1 Advantages

ASLR prevents an attacker from directly referring to objects in memory by randomizing their locations. This stops an attacker from being able to transfer control to his shellcode by hardcoding its address in his exploit.

1.2.2 Limitations

Some attacks on *ASLR* implementations take advantage of the low entropy available for randomization. Where as *ret*2*reg* attacks, allow the attacker to transfer control to their payload by utilizing pointers leaked in registers or memory.

1.3 Return Oriented Programming

ROP is a generalization of the $return - to - libc$ attack. In a $return - to - libc$ attack, the attacker reuses entire functions from *libc*. With *ROP*, the attacker uses instruction sequences found in memory, called *gadgets*, and chains them together. *ROP* attacks are desirable because they allow the attacker to perform computations beyond the functions of *libc* (or whatever code is nonrandomized). This is especially important in the context of modern systems, because the nonrandomized code may not contain useful functions for the attacker.

Chapter 2

Related Work

2.1 Schacham et al

The motivation for the *ROP* attack was *The geometry of innocent of innocent flesh of a bone* which describes $return - into - libc$ attack to be mounted on $x86$ executables that calls no functions at all. The research work describes attack which combines a large number of short instruction sequences to build gadgets that allow arbitrary computation. This paper has mentioned how to discover such instruction sequences by means of static analysis. *Schacham et al* introduced *ROP* technique by which an attacker can induce arbitrary behavior in a program whose control flow can be diverted without injecting any code. A return-oriented program chains together short instruction sequences already present in a program's address space, each of which ends in a *return* instruction. This technique could eventually evade $W \oplus X$-style hardware protections of $x86$ machines.

2.2 Erik Buchanan et al

When good instructions go bad : Generalizing ROP into RISC, showed a generalized approach for return-oriented programming on the $SPARC$, a fixed instruction length $RISC$ architecture with structured control flow. It was the first research work to construct a Turing-complete library of code gadgets using snippets of the Solaris *libc*, a general purpose programming language, and a compiler for constructing return-oriented exploits.

2.3 Dullien et al

A framework for automated architecture-independent gadget search, introduced a framework of algorithms capable of locating a Turing-complete gadget set. The framework uses *Reverse Engineering Intermediate Language* (REIL), machine code into an intermediate language which allows the framework to be used for many different CPU architectures with minimal architecture-dependent adjustments.

2.4 Ralf Hund et al

The first automated *Turing complete ROP-rootkit for Bypassing Kernel Code Integrity Protection Mechanisms* for Windows platform was described in this research work by *Ralf Hund et al.* The framework is partitioned into three core components: *Constructor* for gadget discovery, *Compiler* provides a comparatively high-level language for programming in a return oriented way (converts the source binary to an intermediate high-level language to produce the final memory image of the program), Loader resolves the relative memory address to absolute address as the output of Compiler is position independent. *Q* follows the similar structure in the system design.

2.5 Binary Analysis Platform

Binary Analysis Platform (BAP) makes it easy to analyze binary code by first lifting assembly instructions into a simple language called *BIL* (the BAP Intermediate Language). Unlike assembly, *BIL* only has a few language constructs, which makes it easy to analyze. The goal of *BAP* is to make it easy to develop binary analysis techniques and tools. Which makes it perfect to build the higher analysis system on top of this. *Q* is build on top of *BAP*.

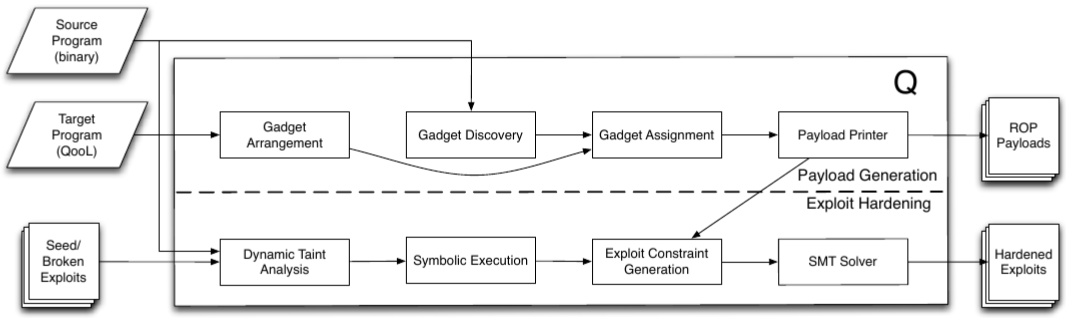
Chapter 3

Contributions

- Proof to show that existing $ASLR$ and $W \oplus X$ implementations do not provide adequate protection by developing automated techniques to bypass them.
- New ideas to scale ROP to small code bases. ROP techniques for small, nonrandomized code bases as found in most practical exploit settings.
- Evaluation of the techniques in an end-to-end system, proving the insufficient security of the existing defenses.
- Semantic gadget arrangement used in this research work is one of the novel technique. Efficiency of Q is more as randomized testing is enabled.

Chapter 4

Summary



Source program: Vulnerable program

Target program: *Q*'s high level language *QooL*

4.1 Gadget Discovery

Finding gadgets in the vulnerable program using semantic program verification techniques. *Q* requires each gadget to satisfy four properties:

1. **Functional**: Each gadget has a type that defines its function. A gadget's type is specified semantically by a boolean predicate that must always be true after executing the gadget.

2. **Control Preserving**: Each gadget must be capable of transferring control to another gadget. This means that the gadget must end with *ret* or some semantically equivalent instruction sequence.

3. **Known-Side-effects**: The gadget must not have unknown side-effects. Eg: The gadget must not write to any undesired memory locations

4. **Constant Stack Offset**: Most gadget types require the stack pointer to increase by a constant offset after each execution

4.1.1 Gadget Types

The set of gadget types in Q defines a new instruction set architecture (ISA) in which each gadget type functions as an instruction. For each and every gadgets a postcondition β is defined which must hold true after the execution of it. An instruction sequence I satisfies a postcondition β if and only if the post condition is true after running I from any starting state.

Name	Input	Parameters	Semantic Definition
NoOpG	—	—	Does not change memory or registers
JumpG	AddrReg	Offset	**EIP** $\leftarrow$ **AddrReg + Offset**
MoveRegG	InReg, OutReg	—	**OutReg** $\leftarrow$ **InReg**
LoadConstG	OutReg, Value	—	**OutReg** $\leftarrow$ **Value**
ArithmeticG	InReg1, InReg2, OutReg	$\Diamond_b$	**OutReg** $\leftarrow$ **InReg1** $\Diamond_b$ **InReg2**
LoadMemG	AddrReg, OutReg	# Bytes, Offset	**OutReg** $\leftarrow$ M[**AddrReg + Offset**]
StoreMemG	AddrReg, InReg	# Bytes, Offset	M[**AddrReg + Offset**] $\leftarrow$ **InReg**
ArithmeticLoadG	OutReg, AddrReg	# Bytes, Offset, $\Diamond_b$	**OutReg** $\Diamond_b\leftarrow$ M[**AddrReg + Offset**]
ArithmeticStoreG	InReg, AddrReg	# Bytes, Offset, $\Diamond_b$	M[**AddrReg + Offset**] $\Diamond_b\leftarrow$ **InReg**

4.1.2 Semantic Analysis

Given I is an instruction sequence and β is the semantic definition, a program verification technique called weakest precondition describes when I will terminate in a state satisfying β, $WP(I, \beta) \equiv true$. Eg: $movl$ 0xc($\%eax$), $\%ebx$; ret is a $LOADMEMG$ gadget. Q converts this to $final(\%ebx) = initial(M[\%eax + 12])$.

4.1.3 Algorithms

Two algorithms are used for gadget discovery. The first algorithm tests using concrete randomized inputs to check whether the semantics of an instruction sequence matches those of any gadget types and validity check of the weakest precondition.

Algorithm 1 Automatically test an instruction sequence $\mathcal{I}$ for gadgets

```
Input: $\mathcal{I}$, $numRuns$, $gadgetTypes[]$
for $i = 1$ to $numRuns$ do
    $outState[i] \leftarrow \mathcal{I}(Random\, input)$
end for
for $gtype \in gadgetTypes$ do
    $\mathcal{B} \leftarrow postconditions[gtype]$
    $consistent \leftarrow true$
    for $j = 1$ to $numRuns$ do
        if $\mathcal{B}(outState[j]) \equiv false$ then
            $consistent \leftarrow false$
        end if
    end for
    if $consistent = true$ then {Possibly a gadget of type $gtype$}
        $F \leftarrow wp(\mathcal{I}, \mathcal{B})$
        if $decisionProc(F \equiv true) = Valid$ then
            output {Output gadget $\mathcal{I}$ as type $gtype$}
        end if
    end if
end for
```

The second algorithm iterates over the executable bytes of the source program, disassembles them, checks for the presence of *ret* instruction and if found then calls first algorithm as a subroutine.

```
Algorithm GALILEO:
    create a node, root, representing the ret instruction;
    place root in the trie;
    for pos from 1 to textseg_len do:
        if the byte at pos is c3, i.e., a ret instruction, then:
            call BUILDFROM(pos, root).
```

4.2 Gadget Arrangement

Gadget arrangement is a way of implementing target program using different types of gadgets discovered from the vulnerable program. One of the major achievement of this research work is that if the most natural choice of gadget is not available, Q effectively tries to synthesize a combination of other gadgets that will have the same semantics due to their efficient semantic engine.

4.2.1 Q's Language: QooL

Enables users to easily interact with the exploited program's environment. *QooL* is not turing complete.

4.2.2 Arrangements

A gadget arrangement is a tree in which vertices are gadget types and edge represents type of input to the next gadget from the output of the previous one. Gadget arrangement is done using maximal munch or longest matching algorithm. The algorithm considers the principle that when creating some arrangement, as much of the available input as possible should be consumed. It also assumes that any instruction selected as the best will always be available for use. A robust gadget arrangement algorithm cannot make assumptions. Q employs every munch which builds tree representing all possible ways that gadget types can be arranged to perform a computation. The munch rules are applied recursively to the program being compiled in Q.

4.2.3 Munch rules

In real case scenarios, many binaries do not contain gadgets for directly storing to memory. If Q is able to set the value in memory to 0 or -1, it can use $ARITHMETICSTOREG$ gadget with mathematical identities to write an arbitrary value. An example gadgets from $apt - get$ source file:

```
; Load eax: -1
pop %ebp; ret; xchg %eax, %ebp; ret
; Load ebx: address-0x5e5b3cc4
pop %ebx; pop %ebp; ret
; Write -1
or %al, 0x5e5b3cc4(%ebx); pop %edi;
   pop %ebp; ret
; Load eax: value + 1
pop %ebp; ret; xchg %eax, %ebp; ret
; Load ebp: address-0xf3774ff
pop %ebp; ret
; Add value + 1
add %al,0xf3774ff(%ebp);
   movl $0x85, %dh; ret
```

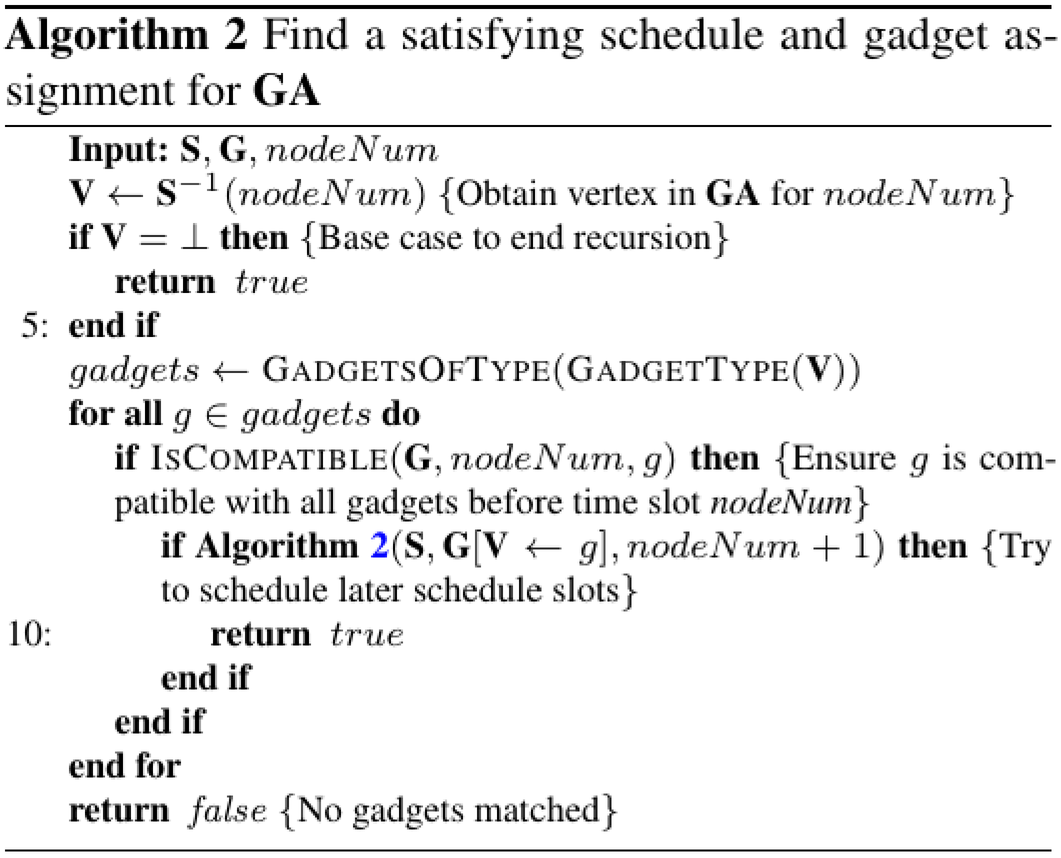
4.3 Gadget Assignment

As long as at least one of the gadget arrangements can be satisfied using the gadgets it discovered in the source program, Q cannot output a working *ROP* payload. The goal is to assign gadgets found during the discovery to the vertices of arrangements and see if the assignments are compatible. If it is successful, the output is the mapping from gadget arrangement vertices to concrete gadgets.

Gadget assignments needs a schedule, as the gadgets must execute in a particular order because there are data dependencies between different gadgets. Hence Q strictly follows the below rules for gadget assignments:

1. **Matching registers**: Two registers should match whenever the result of gadget a is used as input type for gadget b. i.e $OutReg(a) = InReg(b, type)$.
2. **No register Clobbering**: If the output of gadget a is used by gadget b, then a's output register should be not altered by other gadgets which are scheduled in between a and b.

Q's key observation about the gadget arrangement is that if a gadget arrangement **GA** is unsatisfiable, then any **GA'** that contains **GA** as subtree is also unsatisfiable. Gadget assignment is implemented using two algorithms. Algorithm 3 is a caching wrapper which caches results and calls Algorithm 2 which uses brute force technique to go through all the possible gadget assignments on larger subtrees. In-order to reduce the time complexity of the algorithm, it has been designed in such a way that If the algorithm 3 fails on a subtree, it aborts the entire arrangement. Q calls Algorithm 3 on each possible gadget arrangement until one is satisfied or there are none left.



Algorithm 2 Find a satisfying schedule and gadget assignment for **GA**

```
Input: $\mathbf{S}, \mathbf{G}, nodeNum$
$\mathbf{V} \leftarrow \mathbf{S}^{-1}(nodeNum)$ {Obtain vertex in GA for $nodeNum$}
if $\mathbf{V} = \perp$ then {Base case to end recursion}
    return $true$
end if
$gadgets \leftarrow$ GadgetsOfType(GadgetType($\mathbf{V}$))
for all $g \in gadgets$ do
    if IsCompatible($\mathbf{G}, nodeNum, g$) then {Ensure $g$ is compatible with all gadgets before time slot $nodeNum$}
        if Algorithm 2($\mathbf{S}, \mathbf{G}[\mathbf{V} \leftarrow g], nodeNum + 1$) then {Try to schedule later schedule slots}
            return $true$
        end if
    end if
end for
return $false$ {No gadgets matched}
```

Data structures in Algorithm 2:

1. **S**: $V \rightarrow N$; A one to one mapping between each vertex and its position in the current schedule.

2. **G**: $V \rightarrow G$; The current assignment of each vertex to its assigned gadget.

Algorithm 3 Iteratively try to satisfy larger subtrees of a **GA**, caching results over all arrangements.

```
    Input: GA, C
    for all GA' ∈ SUBTREES(GA) do {In order from shortest to tallest}
        if C(GA') = ? then
            C(GA') ← exists S ∈ SCHEDULES(GA') such that Algorithm 2(S, EMPTY, 0) = $true$
5:      end if
        if C(GA') = $false$ then {Stop early if a subtree cannot be satisfied}
            return $false$
        end if
    end for
10: return C(GA) {Return the final value from the cache}
```

Data structures in Algorithm 3:

1. **C**: $V \rightarrow \{0, 1, ?\}$ is a cache which maps a gadget arrangement vertex to one of $true$, $false$ or $unknown$.

4.4 Creating Exploits that Bypass ASLR and DEP

Given an input exploit or even a proof of concept crashing input, Q is capable to provide an exploit which can bypass $ASLR$ and DEP.

4.4.1 Generating formula from a concrete run

At binary level, the recording tool incorporates dynamic taint analysis to keep track of the which instructions deal with the user input or derived data. Q uses this information to keep track of the information to:

1. Record the instruction that access or modifies the taint data.
2. Halt the recording once control-hijacking takes place (i.e when EIP becomes tainted).

After recording the concrete execution, Q symbolically executes the target program, which is similar to normal execution, except that each input byte is replaced with symbol. Such that every computation involving symbolic input is related to a symbolic expression. Constraints on the input to follow the same execution path would be recorded as constraint formula Π. Constraint formula has all inputs that follow the vulnerable path. Q uses two exploit constraints α is set true only if a program's control flow has been diverted and Σ maps to true, only if payload for desired computation is in the exploit. Q uses BAP to convert the binary instructions to an intermediate language for the ease of analysis.

4.4.2 Exploit Constraint Generation

1. **Control Flow Hijacking constraints**: α takes the form $jumpExp = targetExp$, where $jumpExp$ is the symbolic expression representing the target of the jump that tainted the instruction pointer, and $targetExp$ depends on the exploit. For a typical stack exploit, $targetExp = \&(shellcode)$ and for ROP payload, $targetExp = \&(ret)$. This implies that ROP payload must be stored somewhere in the memory.

2. **Computation constraints**: Computation constraints ensure that the computation payload is readily available in the memory in a known address at the time of control flow hijacking. Computation constraints take the form $\Sigma = (mem[payloadBase] = payload[0] \bigwedge \ldots \bigwedge mem[payloadBase + n] = payload[n])$, where $payloadBase$ denotes the starting address of the payload in memory, and payload denotes the bytes in the payload.

4.5 Implementation details

1. ROP component of Q is built on top of Binary Analysis Platform. BAP is used to:

   (a) Record assembly instructions into BAP intermediate language.

   (b) Symbolically execute the trace, to obtain constraint formula Π.

   (c) Compute α and Σ.

2. ML code for gadget discovery, gadget arrangement and assignment phases

3. Dynamic Taint analysis using Pin framework; tracing tool is optimized to only record instructions that are user derived

4. Simple Theorem Prover is used to determine the validity of generated weakest preconditions and to find a satisfying answer to the resulting constraint formula, uses the result to build exploit.

4.6 Evaluation

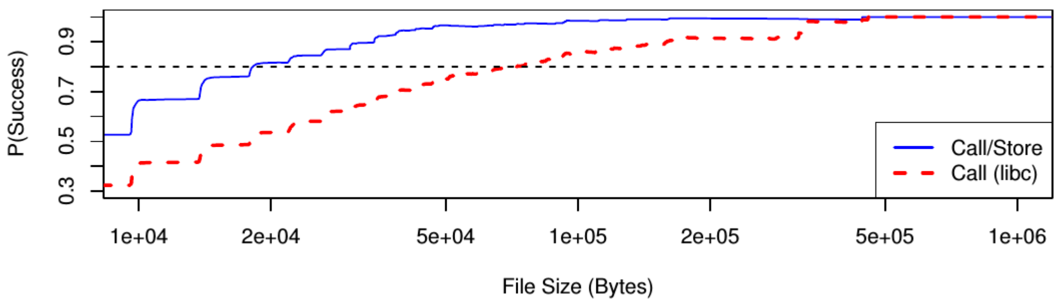
4.6.1 Applicability

Tested on 1,298 *ELF* files in Ubuntu 9.10 $x86$ machine. For each program P, the research work has considered if Q can create a *ROP* payload to:

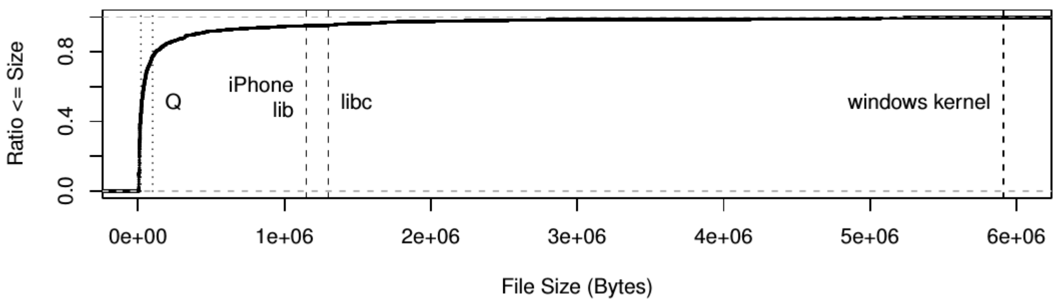
1. **Call functions also called by P**: External functions called by P have an entry in the program's Procedure Linkage Table(PLT).
2. **Call external functions in libc**: Calling external functions that do not have a *PLT* entry is more complicated. For this, a technique for calculating the address of functions in *libc* even when *libc* is randomized. This involves more computation than the above case, and so is more likely to be unsatisfiable.
3. **Write to memory**: A payload which overwrites an arbitrary address

4.6.2 Results

Previous research works shows that *ROP* is more difficult when there is less binary code (¡ 100 *KB*). But the Q has proved that it can call linked functions in 80% of programs that are 20*KB* or larger, and can call any function in linked shared libraries in 80% of programs that are at least 100*KB* in size.



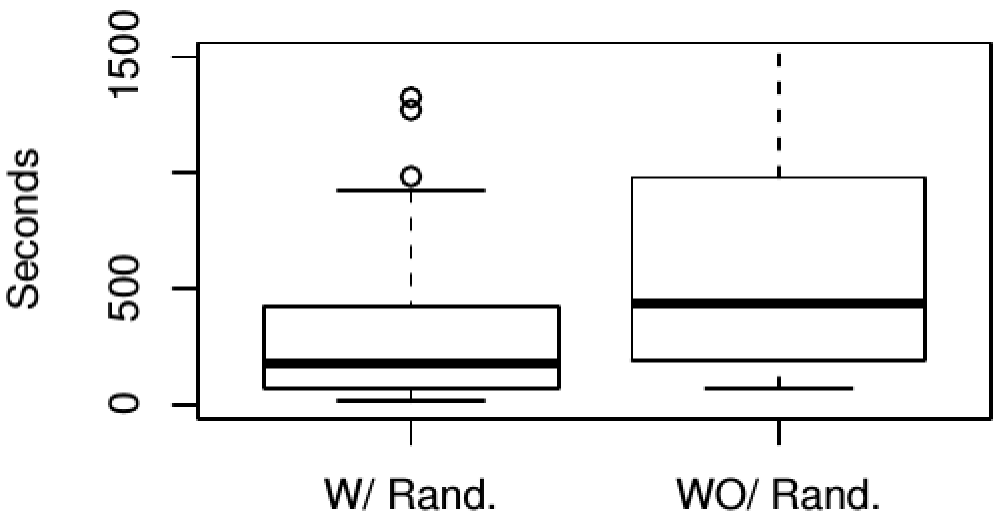
(*The probability that Q can generate various payload types*)



(*The difference in the probability of generating various payload types in other related works and Q with respect to the size of binaries*)

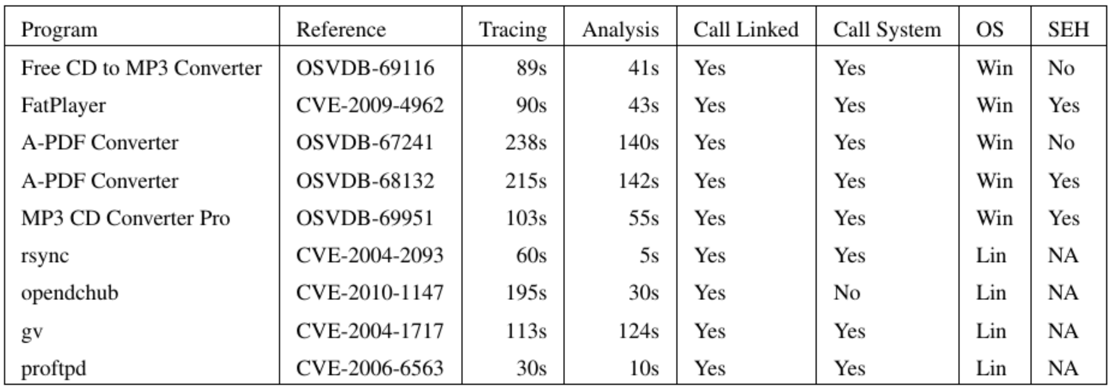
4.6.3 Efficiency

Semantic gadget arrangement is one of the novel technique used in this research work seems to be inefficient if used in a nonrandomized testing cases. Their results shows that Q runs faster when randomized testing is enabled.



4.6.4 Exploit Hardening

The research team has tested exploit hardening capability of the Q system on already available public exploits of some GNU/Linux and Windows programs which are able to exploit only when $W \oplus X$ and $ASLR$ are disabled. The results shows that Q system was able to harden exploits for several large, real program with the current security defences.



Program	Reference	Tracing	Analysis	Call Linked	Call System	OS	SEH
Free CD to MP3 Converter	OSVDB-69116	89s	41s	Yes	Yes	Win	No
FatPlayer	CVE-2009-4962	90s	43s	Yes	Yes	Win	Yes
A-PDF Converter	OSVDB-67241	238s	140s	Yes	Yes	Win	No
A-PDF Converter	OSVDB-68132	215s	142s	Yes	Yes	Win	Yes
MP3 CD Converter Pro	OSVDB-69951	103s	55s	Yes	Yes	Win	Yes
rsync	CVE-2004-2093	60s	5s	Yes	Yes	Lin	NA
opendchub	CVE-2010-1147	195s	30s	Yes	No	Lin	NA
gv	CVE-2004-1717	113s	124s	Yes	Yes	Lin	NA
proftpd	CVE-2006-6563	30s	10s	Yes	Yes	Lin	NA

(*A list of public exploits hardened by Q*)

Chapter 5

Conclusion

5.1 Conclusion

The research work is able to develop a unique technique to synthesize robust ROP exploits which could bypass the current ROP defences including $W \oplus X$ and $ASLR$ for binaries sized above $20KB$ with a success rate of 80%. Q is one of the research outcome, which could be used to harden exploit in-order to bypass OS defences. This research work points out dangers in the current operating systems which doesn't have complete randomized code which leads to the bypass of ROP defences when there is a vulnerability in the programs.

Chapter 6

Possible Future Work

6.1 Possible future work

1. Automatically construct *ROP* exploit payloads that do not use *ret* instructions.
2. An advanced memory analysis that can statically detect when a memory access will be safe, which will allow Q to use more gadgets.