Low-Level Control-Flow Manipulation Techniques

Horia V. Corcalciuc
Computational Physics and Information Technologies
Horia Hulubei National Institute for R&D in Physics and Nuclear Engineering (IFIN-HH)
Str. Reactorului no.30, P.O.BOX MG-6, Bucharest - Magurele, ROMANIA
horia.corcalciuc@nipne.ro

***Abstract*—Contrary to interacting with software remotely at runtime, commercial software may be altered by an attacker that is able to to change the meaning of the program at will by decompiling and recompiling the program. An attacker is able to coerce control-flow, manipulate predicates in order to lead the program into a favourable state. The aim of this paper is to present a strictly limited set of low-level attack patterns and to draw a parallel to exploits carried out against system software that cannot be tampered with.**

***Index Terms*—security, software protection, predicate logic, control flow, UML**

I. Introduction

Software vulnerabilities pertain mainly to runtime attacks by exploiting software that cannot be altered and usually with the purpose of gaining access to a system. This is the case of the taxonomies that study attack patterns "Top Ten Vulnerabilities", "Seven Deadly Sins" or "Pernicious Kingdoms" [1] or time and state attacks [2]. The latter taxonomy focuses on attacks that invalidate or coerce intermediary predicates such that the program enters a state that is favourable to the attacker. The question arises whether runtime attacks and copy protection attacks are related and to what extent. Buffer overflow techniques have been used to defeat software protections, by injecting code early on in the boot process of tablet devices for the purpose of defeating firmware protections [3], [4] and illustrate attacks on software where code can only be minimally altered at runtime. Whilst CERT and "mitre.org's" CVEs explore vulnerabilities with the purpose of compromising system security, there is no classification of attack patterns or vulnerabilities that could compromise software protection.

Concepts such as Aczel traces [5], predicates, UML-style [6] swimlanes and program refinement [7] are used as the means to analyse and illustrate copy protection vulnerabilities. Section II, "Background" presents a short introduction to the technical concepts involving traces, swimlanes and rely-and-guarantee [8]. In the "Techniques" section III, four typical attack patterns will be presented that will then be applied to various commercial software packages. The paper summarises the findings in the "Conclusions" section IV lists the software packages that have been found to vulnerable to the typical attack patterns presented in this paper.

II. Background

A trace [5] represents a state transition from a state justified by a precondition, to a following state justified by a postcondition [9]. Internally, a trace contains one or more state pairs of the form (σ, σ') where σ is the start state and σ' is the end state. Depending on the segment of code that has to be analysed, a trace may contain multiple state pairs (σ, σ'), such that the scope of the analysis can be made larger or smaller. In order for a program to transition from a state that satisfies a precondition $s \models P$ to a state that satisfies a postcondition $s' \models Q$ reliably, we write that $P \triangleright Q \subseteq R$ where R represents the rely which is a set of conditions such that the transition is stable. In turn, a trace offers a guarantee G to the environment which is a set of conditions that will be respected during the state change. In other words, the rely set R would contain the necessary conditions that a program relies on in order to make a transition between states whereas the guarantee set G contains the conditions that the program offers to the environment for the duration of the state change.

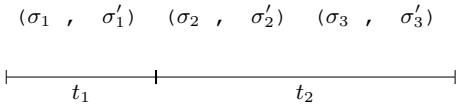

Fig. 1: The composition of two traces t_1 containing the state pairs (σ_1, σ_1') and t_2 containing the state pairs (σ_2, σ_2') and (σ_3, σ_3'). Note that a trace may contain one or more state pairs which allows focusing on small or larger code segments when analysing programs.

A trace t can be composed of multiple traces $t = t_1 \cdot t_2 \cdots t_n$. When composing multiple traces together, the notation $t = t \cdot \checkmark$ is used to indicate that the trace t is a terminating trace which implies that the intermediary state changes have completed successfully.

In practice, a vulnerability in a software package is usually traced to a few instructions such that a fine-grained approach would allow reasoning only about the relevant code segments.

Since software security deals with the cause rather than the effect of a flaw, it is only possible to offer a specification - for instance, by specifying the set of relies and guarantees that traces have to respect in order to ensure that a program's internal state changes are legitimate.

A. Swimlanes

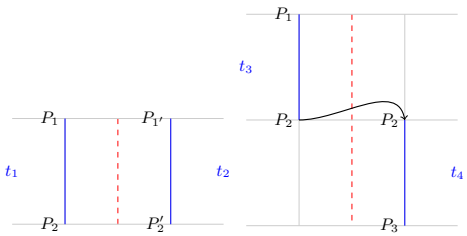

Fig. 2: On the left two traces t_1 and t_2 run concurrently to each other on parallel swimlanes. On the right, two traces t_3 and t_4 run sequentially where t_4 follows after t_3. Pre- and postconditions are illustrated at the beginning, respectively the end of each terminating trace. The trust boundaries indicated with a dashed line delimits the trust between the two traces.

Swimlanes (Figure 2) are an UML-style [6] depiction of control flow [10] that can be used to illustrate the branching and concurrent threads or processes. Contrasted with extensions on UML such as UMLsec [11], swimlane diagrams are based on traces and can

```
; read
mov    qword [ss:rbp+0xfffffffffffffd0], rax
; compare
cmp    byte [ds:UpdateRegistrationData], 0x0
; jump
jle    0x1000d2c1e
; continue
mov    edi, dword [ds:objc_msg_alloc]
```

Fig. 3: Disassembly of a conditional statement in a program that illustrates the typical *read*, *compare* and *jump* operations performed at runtime.

provide a fine-grain examination of control-flow by illustrating state-transitions as well as the pre- and post conditions governing them.

On each swimlane, a trace can be placed that is bounded at the top by a precondition satisfying a begin state and ends with a postcondition satisfying an end state. Stability can be depicted next to the trace, showing the rely that the trace depends on and the guarantee that it offers to the environment.

III. Techniques

Assembler is used to illustrate control-flow using the smallest discernable building blocks. The minimal attack patterns discussed in this paper can be summarised as:

- Replace a conditional jump (`jne`, `je`, etc...) by an unconditional jump (Figure 4), ex: `jmp` or invert a conditional, for instance a "jump if not equals" (`jne`) replaced by a "jump if equal" (`je`).
- Eliminate a jump altogether, for example, by replacing a conditional or an unconditional jump by a no-operation (`nop`) instruction (Figure 5) such that the control flow passes directly to the following instruction.
- Remove a function entirely by executing an unconditional jump from the entry point of a function to the end of the function (Figure 8).
- Build a `nop` sledge (Figure 10) from one instruction to a different instruction by adding multiple `nop` instructions sequentially, thereby "carrying" a predicate to a different instruction.

Using just four basic attack patterns, a surprisingly large amount of software packages have been made to believe that they either are registered versions, that the trial limitations have been completely removed or that the trial time allotted to a demonstration version has never elapsed.

A. Manipulating Jump Instructions

On a low level, a conditional statement can be generalised to the sequence of operations: *read*, *compare* and *jump*, where *read* places a value into a register, *compare* compares that value to the value of another register and the *jump* is an explicit divert of control-flow based on the outcome of the *compare* operation. If a conditional statement would be disassembled then the result would end up as code pattern similar to the one presented in Figure 3.

The code presented in Figure 3 can be depicted graphically using swimlane diagrams (Figure 4). The original implication denoted by the stability condition $P_2 \triangleright P_2 \vee P_3'$ is that the postcondition P_2 of the compare operation in trace t_2 would either lead to P_2 or to P_3' both of which would have been legitimate control flow transfers. By injecting an unconditional jump using a `jmp` instruction the postcondition of the compare operation `cmp` is ignored such that regardless of the outcome the branch leading to P_2 is always favoured. The transition is still legitimate, because P_2 was a possible outcome but due to the injection leading to P_3' becomes impossible.

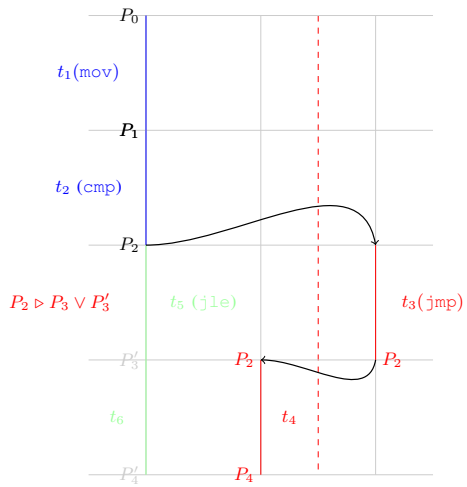

Fig. 4: Representing control-flow graphically as a program executes a series of commands, represented as traces $t_{1..5}$ and branches on a postcondition P_2 that may favour one branch instead of the other depending on the instruction used in the antecedent traces.

In terms of code, this can be understood as a situation where the software would perform a check to determine if the software is registered or not, and then depending on the outcome, some limitations may be removed, respectively imposed.

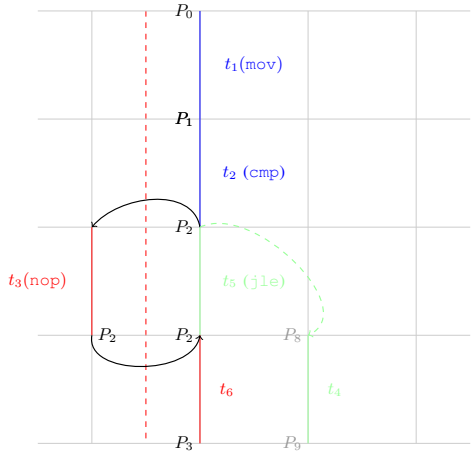

Fig. 5: Whilst the program has two possible outcomes after performing the check in t_2 using a compare operation `cmp` there are two branches possibly selected. The `jle` instruction leads either to the postcondition P_8 following the trace t_5 or to P_2. Injecting a `nop` operation changes the meaning of the `jle` instruction and the program is coerced to always lead to P_2.

In a practical scenario, assuming that the described attack is performed on some software that depends on the compare operation to determine whether or not it is registered, the outcome of the test would have been coerced by the attacker to invariably lead to the branch that corresponds to the software being registered. Conversely, should the software at a later time be registered, then this attack will in fact make the software unregistered due to the switch of meaning of the conditional jump from `je` to `jnle`.

The converse operation of injecting an unconditional jump via `jmp` is to inject `nop` operations (Figure 5) such that control-flow will never be diverted. In both instances, both an unconditional jump `jmp` and

```
; read
mov    qword [ss:rbp+0xfffffffffffffd0], rax
; compare
cmp    byte [ds:UpdateRegistrationData], 0x0
; jump
jmp    0x1000d2c1e
nop
; continue
mov    edi, dword [ds:objc_msg_alloc]
```

Fig. 6: Contrasted with Figure 3, the `jle` has been overwritten by an unconditional jump instruction `jmp` that will make the program ignore the rest of the instructions and forcibly jump to another part of code.

```
; Basic Block Input Regs: rsp
; methImpl_AppController_demoExpired:
100003419    push    rbp
10000341a    mov     rbp, rsp
10000341d    jmp     0x100003428
; ...
100003428    pop     rbp
100003429    ret
```

Fig. 7: Eliminating a function by jumping over all the instructions using an unconditional jump `jmp`.

a no-operation `nop` will "carry" the precondition P_2 and select one or the other branch whereas flipping the meaning of a conditional jump will favour either one of the branches.

B. Code Elimination

Another low-level attack technique can be used in cases where a function is responsible for say, disabling the application if the trial has expired, and is carried out by eliminating the function altogether. The attack implies jumping from the entry-point of a procedure to the end of the procedure, thereby skipping all instructions. A commercial software named "ShareMouse" relies on a timer that is installed and periodically triggers in order to check whether a certain amount of time has elapsed and disables itself in order to prevent further usage. The application then has to be restarted in order to be able to continue using the program. A practical attack on "ShareMouse" involves eliminating the callback function of the timer thereby making the software usable beyond the time limit (Figure 7).

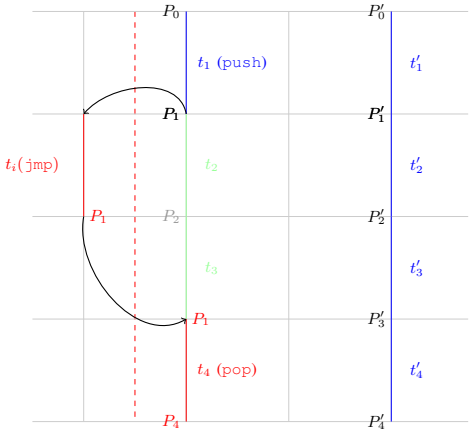

Fig. 8: A depiction of a slice of a program where the traces $t'_{1\ldots5}$ represent the main thread executing some code, the traces $t_{1\ldots5}$ represent a timer callback that runs periodically and t_i is an instruction injected by the attacker.

```
00022b54    beq         0x22b80
; ...
00022b58    nop
; ...
00022b7c    nop
00022b80    ldr         r0, = 0xd12f8
```

Fig. 9: An example taken from the PocketPC "PocketRSS" software package where control flow is passed along a series of `nop` instructions, forming a `nop` sledge, from one part of code to a different part of code whilst preserving the initial state.

In Figure 8 a process executes code in parallel on two different threads, illustrated on two different simlanes, where the traces $t'_{1\ldots4}$ run on the main thread, the traces $t_{1\ldots4}$ run in a different thread and t_i is a trace injected by the attacker. From the swimlane diagram we see that the callback function executing in parallel to the main thread is a composition of traces $t = t_1 \cdot t_2 \cdot t_3 \cdot t_4$ where trace t_4 is a terminating trace such that $t_4 = t_4 \cdot \checkmark$. Due to the injected trace, the callback function becomes a different composition of traces $t = t_1 \cdot t_i \cdot t_4$ where t_i is the trace injected by the attacker. It is sufficient for the attacker to ensure that the last trace t_4 is a terminating trace ($t_4 = t_4 \cdot \checkmark$) in order for the attack to succeed. Multiple stability conditions could be used to ensure that the legitimate traces execute, formulated as $P_1 \triangleright P_2$ and $P_2 \triangleright P_3$ where P_3 would have been the legitimate precondition for the trace t_4 in Figure 8.

There are cases when regions of code are protected by canaries [12], where the injection of a trace t_i would trigger the protections such that t_4 would not be a terminating trace and would result in an exception being thrown [13]. The attacker must make sure that by diverging control-flow, the state to which the injected trace leads to, will still satisfy the post-condition P_1.

C. Trivial Control Transfer

The act of "carrying" a predicate such as a pre- or postcondition is closely related to `nop` sledge attacks (Figure 9). In a typical `nop` sledge attack, the control-flow of a program is manipulated in such a way that the outcome of an operation (in effect, a post-condition to a trace) is "dragged" to a different region of code where some decision is made upon the original outcome. This is equivalent to overwriting code using `nop` instructions from the point where a precondition has to be carried and up to the desired instruction. A similarity can be drawn to "Time of Check to Time of Use" (TOCTTOU) attacks [14], [15] where, by contrast, a set postcondition is forced to be still valid at a later time when it will be used.

Many software packages typically perform checks when the program is launched and either refuse to execute, display a nag screen or instruct the user to buy the fully registered version in order to be able to continue using the program. To that effect, the main entry routine for a program becomes a good location to probe for vulnerabilities. In fact, a lot of software that was analysed, performed various registration checks in the body of the procedures `applicationDidFinishLaunching` or `applicationWillFinishLaunching` that are called by specification during program startup.

In Figure 10 a segment of code is skipped entirely by forming a `nop` sledge that lets execution skip the legitimate intermediary traces t_2 and t_3. Although the legitimate control flow could be described as a composition of traces $t = t_1 \cdot t_2 \cdot t_3 \cdot t_4$ where t_4 is a terminating trace $t_4 = t_4 \cdot \checkmark$ the injection of the traces t'_1 and t'_2 changes the meaning of the program such that the composition of traces $t = t_1 \cdot t'_1 \cdot t'_2 \cdot t_4$ is obtained instead. Similarly to eliminating functions altogether, the

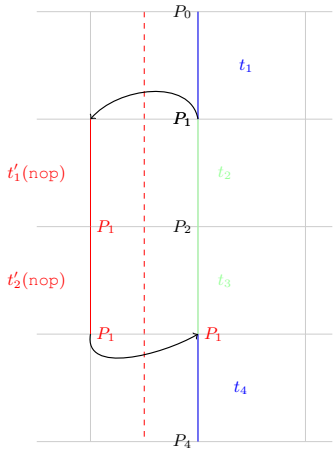

Fig. 10: A code segment where the traces $t_{1\ldots4}$ represent the unaltered program and the traces $t'_{1\ldots2}$ being injected by the attacker over the trust boundary. The effect is that the legitimate traces t_2 and t_3 are skipped thereby carrying the post-condition P_1 to the last trace t_4.

attacker must only ensure that the last trace is a terminating trace $t_4 = t_4\checkmark$ in order for the attack to be legitimate.

It is observable that injecting a single unconditional jump instruction (`jmp`) would have achieved the same effect. Depending on the environment or constraints imposed on the injected code, one may chose to use either a `jmp` or build a `nop` sledge.

IV. Conclusions

Software Name	Protection	Used Techniques
Comic Life (v3.x)	Nag screens and popups.	Function elimination.
Folx (v4.x)	Nag screens and trial-only restricted features.	Inverting jumps, `nop` sledges.
Jitouch (v1.x and v2.x)	Time limitations.	Function elimination, `nop` sledges.
Jump Desktop (v5.x)	Nag screen, interface limitations and copy protections.	`nop` sledges.
Keyboard Maestro (v6.x)	Registration required.	Inverting a single jump.
Lyn (v6.x)	Time limitation and missing features.	Inverting jumps.
PhotoSweeper X (v2.x)	Nag popups and time limitations.	Inverting jumps, `nop` sledges.
PocketRSS (v2.1.7)	Time limitations.	`nop` sledges.
Scrivener (v2.6 and v2.7)	Time limitations and nag screens.	Inverting jumps.
ShareMouse (v1.x, v2.x and v3.x)	Time limitations and missing features.	Inverting jumps, `nop` sledges, function elimination.
SideKick (v4.2.1)	Nag screens and time limitations.	Function elimination.
uTorrent (v1.7.x and v1.8.x)	Adverts and spam.	Inverting jumps.

Fig. 11: A presentation of software that spans across several versions and is vulnerable to one or more attack techniques presented in the "Techniques" III section.

The table in Figure 11 summarises the results by providing the name of the vulnerable software, the protection it had originally and the techniques used in order to defeat the protections. A full list would include other software such as: "Acorn", "Alfred", "Amnesty", "BBEdit", "CleanGenius", "CornerStone", "DaisyDisk", "Decloner", "DropDMG", "Entropy", "FontAgent Pro", "Grappler", "iGlasses", "IconBox", "iPulse", "iRamDisk", "Latexian", "Leech", "Omnigraph Sketcher", "Omni Plan", "On The Job", "PathFinder", "Perfect Photo Suite", "PhotoSweeper", "ProxyCap", "QPict", "SecuritySpy", "Smasher", "SnapzProX", "Snippets", "SubethaEdit", "TextMate", "Transmit", "VelaClock", "WireTap Studio", "XScope" and "Zoom2" that have been left out for brevity. All of these software packages were attacked using different combinations of just the four attack patterns presented in the "Techniques" section III and lead to a fully-working version of the software with all features unlocked.

Real-life attacks on copy protection are far more elaborate, sometimes even involving hardware attack vectors, but it is remarkable to observe how a minimal set of carefully picked techniques can alter the entire outcome of a program. Furthermore, since many of the described attacks involve minimal changes to the code, it would be interesting to study cases where binary data could be modified in-flight via a Man-In-The-Middle (MITM) attack in order to change the behaviour of software as it is transmitted over the wire.

V. Acknowledgements

The authors would like to thank Dr. Minhea Dulea of the Computational Physics and Information Technologies department at IFIN-HH for continuous support and would like to mention that the work summarized in this paper was funded by the Ministry of Research and Innovation through project PN18 09 02 05.

References

[1] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[2] Horia V. Corcalciuc. A taxonomy of time and state attacks. *International Journal of Secure Software Engineering (IJSSE)*, pages 40–66, 2013.

[3] Sebastian Höbarth and Rene Mayrhofer. A framework for on-device privilege escalation exploit execution on android. *Proceedings of IWSSI/SPMU*, 2011.

[4] Cedric Halbronn and Jean Sigwald. iphone security model & vulnerabilities. In *Proceedings of Hack in the box sec-conference. Kuala Lumpur, Malaysia*, 2010.

[5] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency verification: introduction to compositional and noncompositional methods*. Cambridge University Press, New York, NY, USA, 2001.

[6] Martin Fowler and Kendall Scott. *UML distilled: applying the standard object modeling language*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1997.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[8] C Viktor Vafeiadis, Viktor Vafeiadis, Viktor Vafeiadis, Matthew Parkinson, Matthew Parkinson, and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *In 18th CONCUR*, pages 256–271. Springer, 2007.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[10] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13:135–152, 2000. 10.1023/A:1010026413531.

[11] Jan Jürjens. Umlsec: Extending uml for secure systems development. In *UML: The Unified Modeling Language*, pages 412–425. Springer, 2002.

[12] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo*, 1999.

[13] Christophe Dony. Exception handling and object-oriented programming: towards a synthesis. *SIGPLAN Not.*, 25(10):322–330, September 1990.

[14] Jinpeng Wei and Calton Pu. Tocttou vulnerabilities in unix-style file systems: an anatomical study. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[15] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.