Towards Automatic Exploit Generation for Android C Libraries

Bhargava Shastry, Lee SeoungKyou

Rice University
Houston, USA

ABSTRACT

Managing the security of complex systems poses a great challenge. Incremental software patches are by far the most common means of plugging security holes: be it at the level of the operating system kernel, the middleware or the applications. However, the practice of system hardening via patch deployment assumes that the concerned patches are deployed uniformly across systems running a piece of software. This is far from reality: software vendors often stagger the deployment of patches for various reasons. This project is a step in the direction of building a static analysis framework that is able to analyse differences in code across multiple versions of a component in a large software system and provide as output, inputs to these components that might compromise security of the system. We choose the Android mobile platform as a case study where our framework could be put to test.

1. INTRODUCTION

Contrary to the view that security patches enhance the security of a software system, recent work [3]—henceforth refered to as APEG —shows how the former can actually aid attackers in exploiting the very same vulnerabilities that were patched by exploiting the vulnerability on the unpatched version of the program. Note that this is nontrivial: a security patch need not reveal the vulnerability in a straightforward way; on average, it would take a seasoned hacker a few days of work at the very least to write an exploit to undermine an unpatched program. One would need to analyse the target program to list program variables that are effected by the patch and where they are used and ultimately craft an input that would attack the unpatched program.

A good example of staggered patch deployment is the Android mobile platform, where, due to fragmented variants of the platform (hardware and software), different phones seem to be running different versions of the Android operating system. The extent of fragmentation has been reported by OpenSignal [7]. It has to be noted that not only is the device (hardware) fragmented, but so is the entire software stack being used by a given device; the API-level (Android version) and the underlying Linux kernel version. Android's open nature has led to a proliferation of devices, each with subtly different hardware, running different variants of the software stack, not to talk about custom ROMs that are also available.

How is automatic patch-based exploit generation relavant to Android? The staggered nature of patch deployment is more of a problem for an open platform like Android than other closed mobile platforms like Apple's iOS. Android's customisability is a double-edged sword—while users get to choose from multiple varied devices, app developers need to cater their offerings to each of these devices. More importantly, from a security point-of-view, vendors need to ensure that relevant security patches are deployed uniformly across all devices from time to time. The reality is far from what one desires: **TODO: cite reports here**

2. PROBLEM STATEMENT

Formally, APEG proposes techniques for the automatic patch-based exploit generation problem: given a program P, and a patched version of the same program, P', automatically generate an exploit for the potentially unknown vulnerability in P but patched in P'. Furthermore, the work targets input validation vulnerabilities only: vulnerabilities arising out of invalid inputs to the program. Hence, the attack can be modelled as a constraint satisfaction problem, with the constraint that an input fail the input sanitisation checks added in P' with an assumption that there is an exploit corresponding to a given input validation vulnerability. APEG argues that given that the delivery of patches is staggered over long time periods, there is a realistic time window in which unpatched systems are vulnerable to attack. Furthermore, if exploits could be patch-driven in an automatic fashion, a hypothetical attacker would simply wait for patches to be available and subsequently exploit systems that have not been patched.

In this project, we seek to emulate the work done in [3] in a different setting. More precisely, our setting is different from theirs in the following ways:

1. APEG works on closed-source binaries (executables), while we only consider open-source software. This lets us focus on the problem of automatic exploit generation and not worry about reverse engineering a patch. Specifically, our target of attack would be userspace libraries (written in C language) in the Android software stack e.g., *bionic* libc, dalvik runtime libraries etc.

2. APEG proposes techniques based on dynamic program analysis only, static program analysis only and a mix of both, while we focus on static analysis only.

3. RELATED WORK

While our work itself is based on the problem that the APEG defines, there is a broad literature of prior works in this direction.

3.1 Algorithmic trade-offs

The algorithms used in [3] are derived from previous works [2, 4]. Specifically, [4] notes that constriant formulae derived using the weakest pre-condition for a given vulnerability tend to be compact and faster to solve; this benefit positively influences the number of control-flow paths that could be explored in turn resulting in fewer false negatives compared with other approaches to constraint formula generation.

A key concern in related literature is the tension between soundness (in a logical sense) of the proposed mechnisms and the efficiency with which they operate i.e., time to uncover as many control-flow paths as possible. Logical soundness means that there could be false positives but no false negatives whatsoever. For a static analysis approach to be sound, it needs to enumerate possible control-flow paths exhaustively, thereby leading to a a complexity that is exponential in the number of branches in a given program.

One can compromise on soundness by relaxing the requirement (for soundness) that all paths be explored; by exploring a subset of paths (possibly guided by complementary dynamic analysis), one could bring down the complexity of computing satisfiable constraint formulae from exponential to polynomial.

3.2 Automatic test case generation

Korat [1] is an automatic test case generation tool for Java programs. It relies on an annotated version of a Java method and generates test inputs to the method that satisfy the method's pre-conditions (constraints on input space) and uses the method's post-condition (specification on the method's output) as a test oracle to evaluate the correctness of the method's output for the automatically generated inputs.

A more targeted approach to test case generation is taken by the authors of [6]. In this paper, the problem is not to simply generate test cases for a given (C) program, but to actually generate test cases for those runs of the program in which a selected point in a given procedure is executed; this approach is termed goal-based test generation, the goal generation of inputs that implies execution of a certain control-flow path. The idea is closest to that in [3], where, inputs that fail a known check in the patched program are of interest.

4. BACKGROUND

TODO: Talk of the following things: (1) Theory of program verification; program annotations for verification, pre-condition, post-condition; (2) Control-flow graph and its computation; (3) Approach employed in this project, perhaps a running example.

We will use the running example of a program (Algorithm 1) that computes the n^{th} power of a given integer to introduce the reader to program verification terminology.

Let Algorithm 1 refer to a program P, consisting of statements $S_1, S2, ..., S17$. The pre-condition, *pre*, of program P formalises conditions that are required to be satisfied by the program prior to its execution; in the present example, it is a function of x and n. The post-condition of the program, *post*, formalises conditions that are required to be satisfied by the program after its execution. Note that *pre* and *post* are both expressions in first order logic i.e., a boolean predicate.

Program statements $S_1, ..., S_{17}$ may contain variables ($x, y \in V$), arithmetic and boolean expressions (E), and be composed of assignment statements ($V := E$) , if-then-else statements, and while loops.

Algorithm 1 Calculate $y = x^n$

```
Require: $n \geq 0 \vee x \neq 0$
Ensure: $y = x^n$
 1: $y \Leftarrow 1$
 2: if $n < 0$ then
 3:     $X \Leftarrow 1/x$
 4:     $N \Leftarrow -n$
 5: else
 6:     $X \Leftarrow x$
 7:     $N \Leftarrow n$
 8: end if
 9: while $N \neq 0$ do
10:     if $N$ is even then
11:         $X \Leftarrow X \times X$
12:         $N \Leftarrow N/2$
13:     else {$N$ is odd}
14:         $y \Leftarrow y \times X$
15:         $N \Leftarrow N - 1$
16:     end if
17: end while
```

The triple $pre\{S\}post$ is called a Hoare triple named after C.A.R. Hoare, a pioneer in the field of program verification. The Hoare triple is to be read as follows: If the pre-condition is satisfied by a program P consisting of statements $S_i \in S, i = 1, ..., n$, then, on execution of the statements in S, the post-condition will necessarily be satisfied should the program be correct and assuming that it terminates.

4.1 Weakest preconditions and strongest post conditions

We borrow introductory material and definitions of the weakest pre-condition and strongest post-condition in this section from [5]—a good survey of verification methods used today.

Consider two boolean predicates A and B. If the relation $A \implies B$ is true, then A is said to be stronger than B (since A implies B); conversely B is said to be weaker than A.

The strongest post-condition of a program P with a pre-condition *pre* is denoted by $Sp\{P\}\ pre$. The strongest post-condition is the predicate obtained by transforming the pre-condition according to the interpretation of program statements in P. The strongest post-condition is termed so because, given a predicate *post* that is true of any state obtained by interpreting the program P from an initial state where *pre* holds, $Sp\{P\}\ pre \implies post$.

The weakest pre-condition of a program with a post-condition *post* is denoted by $Wp\{P\}\ post$. The weakest pre-condition is the predicate obtained by transforming the post-condition according to the interpretation of program statements in P in reverse order. The weakest pre-condition is weaker than the *pre* in that $pre \implies Wp\{P\}\ post$.

A relation between the Hoare triple $pre\{S\}post$ for a given program P and the weakest pre-condition Wp and the strongest post-condition Sp is that the Hoare triple holds if and only if $(Sp\ S\ pre) \implies post$ or if and only if $pre \implies (Wp\ S\ post)$.

5. FRAMEWORK

TODO: Talk about what we have built: a framework that is capable of statically analysing C code.

How far are we from what we set out to do? Our framework can currently scan simple C code and infer certain properties (abstractions about program variables) about it. This has to be applied to the problem of exploit generation; specifically generating weakest pre-condition, given a program and a patch. Tools we use: Ocaml: functional language that is type safe and models effects. CIL: Compiler front-end for C programs. Good for static analysis. CIL-template: Sample programs for CIL. Build scripts reused. Our framework for static analysis is modeled for a subset of the C language: it supports simple linear arithmetic—the use of $+$ and $\times$ operator on program variables and constants.

6. DELIVERABLES

One of the main goals of this project was to develop a framework that is capable of performing static analysis on C code. We believe our project takes this goal forward. We have a proof of concept static analysis framework that is capable of inferring simple relationships between program variables. The present framework has several limitations, namely, (1) It works for a limited subset of C language instructions and statements; prominently, pointers are not handled; (2) It does not work well for loops in general; we intend to make static analysis in the presence of loops more precise; (3) Finally, our framework is presently modeled on a simplistic constraint language consisting of boolean operators and the less than or equal to operator, $\leq$; for instance, at each point in a C program, our framework keeps a track of constraints in a first-order logical formula consisting of logical operators $\neg$, $\wedge$, $\vee$, and the $\leq$ arithmetic operator. Furthermore, we confine our analyis to linear arithmetic expressions (combination of $+$ and $\times$ operators on program variables and constants)

7. CONCLUSION AND FUTURE WORK

8. REFERENCES

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[2] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[3] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143 –157, may 2008.

[4] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 311–325, Washington, DC, USA, 2007. IEEE Computer Society.

[5] M. Gordon and H. Collavizza. Forward with Hoare. In A. W. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, chapter 5, pages 101–121. Springer London, London, 2010.

[6] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, Mar. 1998.

[7] OpenSignal. Android fragmentation visualized. `http://opensignal.com/reports/fragmentation.php?`