XIFER: A Software Diversity Tool Against Code-Reuse Attacks

Lucas Davi[1], Alexandra Dmitrienko[2], Stefan Nürnberger[1], Ahmad-Reza Sadeghi[1,2]

[1]Technische Universität Darmstadt, Germany [2]Fraunhofer SIT, Darmstadt, Germany

ABSTRACT

The enormous growth of mobile devices and their app markets has raised many security and privacy concerns. Runtime attacks seem to be a major threat, in particular, code-reuse attacks that do not require any external code injection (e.g., return-to-libc or return-oriented programming).

We present, for the first time, a code transformation tool that completely mitigates code-reuse attacks by applying software diversity to the binary at *runtime*. Our tool *XIFER* (1) randomly diversifies the code of an application over the entire memory for *each* invocation, (2) requires no source code or any static analysis, (3) can be applied to both Intel x86 and ARM Linux executables, and (4) induces a negligible runtime overhead of only 1% in average.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Software Diversity, Runtime Attacks, ARM, x86

1. INTRODUCTION

Smartphones and tablet computers have become extremely popular with steadily growing sales figures. Probably the most important success factor for their popularity is the availability of a vast number of applications, ranging from games over messaging apps to office applications.

The high number of available mobile apps largely increases the attack surface on mobile devices: developers of apps are not necessarily security experts, and hence, apps may suffer from various vulnerabilities that can be exploited to compromise the device and access sensitive information (e.g., contacts, SMS, location). This can be observed by the recent discovery of severe vulnerabilities in mobile applications allowing diverse runtime attacks [6, 7, 9].

In conventional runtime attacks, the adversary injects his own malicious code into the program's memory space. However, data execution prevention (DEP) is an effective countermeasure against these types of attacks. On the other hand, code-reuse attacks do not require code injection; instead they only exploit existing code pieces residing in the program's memory space. For instance, return-into-libc attacks use critical functions of the standard Unix C library.

S3'12, August 26, 2012, Istanbul, Turkey.
ACM 978-1-4503-1528-9/12/08.

More sophisticated and general code-reuse attacks such as return-oriented programming (ROP) [11] chain together small code sequences from different parts of the application code (and its included libraries) to generate arbitrary (malicious) program behavior.

Software Diversity. Cohen presented the idea of *software diversity* (or program evolution) to protect computer systems and their running software programs against software exploits [1]. The basic observation is that an adversary typically generates an attack vector and aims to simultaneously compromise as many systems as possible using the same attack vector (i.e., one attack payload). To mitigate this *ultimate attack*, Cohen proposes to diversify a software program into multiple and different instances while each instance still covers the entire semantics of the root software program. The goal is to force the adversary to tailor a specific attack vector/payload for each software instance, and in this way make the attack tremendously expensive and thus reduce its impact.

Further, Cohen argues [1] that diversification of the binary code is non-trivial and hence advocates complier-based diversification that requires access to the source code. Franz [4] has recently explored the feasibility of a compiler-based approach, and suggests that app store providers integrate a multicompiler (diversifier) in the code production process. However, a compiler-based approach has several shortcomings: First, app store providers have no access to the app source code. This requires the multicompiler to be deployed on the developer site, who has to deliver thousands of app copies to the app store. Second, the proposed scheme requires software update processes to correctly patch app instances, a task that can be highly involved. Finally, the obvious drawback is that the app instance installed on the customer's device remains unchanged until an update is provided, which increases the chance of an adversary compromising this particular instance.

In contrast, the code transformation approach typically aims at rewriting machine instructions. A very recent proposal targeting Intel x86 has been made by Pappas et al. [10]. Although the proposed solution can be applied to stripped binaries, it is static (i.e., it would break the signature of mobile apps) and cannot prevent conventional return-into-libc attacks, because all functions remain at their original position. Further, it requires the distribution of thousands of instances since diversity is not applied at runtime. At the same time, Hiser et al. [5] presented instruction location randomization (ILR), a virtual machine based approach that randomizes the location of each instruction, and guides the

execution according to a so-called fall-through map. However, ILR requires static analysis, and induces a higher performance overhead than [10] or our solution presented in this paper.

Lastly, the well-known memory randomization (ASLR) can be considered as a special case of software diversity [3]. It aims at providing protection against runtime attacks by diversifying the application's memory layout. In particular, ASLR randomizes the base addresses of loaded process sections. However, current ASLR realizations are vulnerable to guessing [12] and disclosure attacks [13], because only the base address of an application is randomized.

Contribution. We present a *runtime* software diversifier for mobile devices (for ARM and x86) that tackles the shortcomings of existing mitigation solutions against code-reuse attacks. Our tool, called *XIFER*, diversifies a program at each run without requiring source code. It transforms the control-flow graph of a program and allows injection and splitting of nodes. In particular, it randomizes small code units, the *basic blocks* (BBLs) in the application's memory space, while their size and entropy is derived from a security parameter. It thereby splits contiguous functions in individual pieces and randomly distributes them in memory, which allows for rendering code-reuse attacks such as return-into-libc and ROP [11] infeasible. This introduces a highly randomized memory layout, and tremendously increases the randomization entropy compared to proposals that only reorder functions [8]. Our solution operates *entirely* at runtime rather than compile- or static analysis time and performs software diversity for each application run.

2. RUNTIME SOFTWARE DIVERSITY

To limit code-reuse attacks against user applications, we apply the concept of *software diversity*, and in particular propose *runtime software diversity* as an efficient and practical mitigation approach. Our diversification is dynamically performed at load-time, thereby leaving digital signatures attached to code intact.

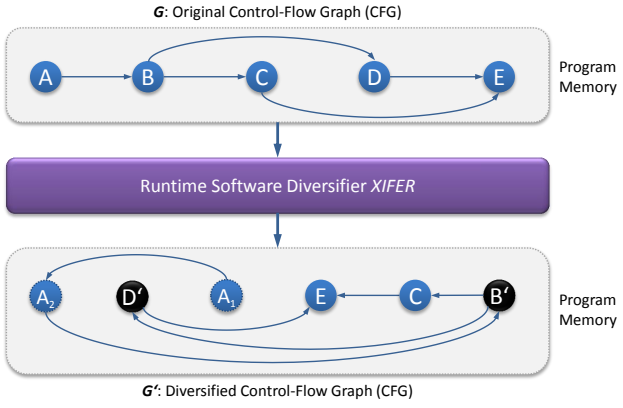

Figure 1: High-Level Idea of *XIFER*

Basically, a mobile app is represented by its corresponding control-flow graph (CFG) which covers all valid execution paths. Each node of the CFG denotes a basic block (BBL), which consists of a sequence of machine instructions with a single entry and exit (e.g., branch instructions). Due to the linear program memory layout, the GFG will be represented flattened in memory when the program is loaded (see Figure 1). After the program has been loaded into the memory, and its signature has been verified, we transform (including random code permutation) the layout of the control-flow graph (CFG) G to G'. The transformed GFG G' is isomorph to G, and hence, covers the entire control flows and semantics of the original CFG G.

As core diversification techniques we leverage BBL *permutation* [2], BBL *splitting* [1], and *injection* of new instructions [1], while the novelty of our approach resides in the fact that we entirely perform the diversification at runtime in memory:

- **Permutation:** We permute BBLs in memory to move them from their original memory position. For instance, in Figure 1, BBL E is moved from the end to the middle of the memory space. Moreover, we distribute BBLs belonging to a single function across the entire memory space which yields a highly diversified memory layout.
- **Splitting:** In addition, we further increase the entropy, by splitting BBLs in multiple BBLs and distributing them (with the same permutation approach as mentioned above) across the memory space. The number of those artificial splittings can be dialed using a security parameter. For instance, BBL A has been split in two BBLs A_1 and A_2.
- **Injection:** Finally, we inject new instructions within a BBL (e.g., BBL B is transformed to BBL B'), and insert new (dummy) BBLs into the application. To preserve the program's semantics, the new code will perform only nop operations.

Note that G' only represents one possible control-flow graph, and the number of possible graph transformations is extremely high. Hence, our approach overcomes the shortcomings of all existing ASLR-based mechanisms: only for the BBL permutation we already achieve an entropy of $n!$, while n denotes the number of BBLs within an application. This means that for a sample 40kb binary with around 250 BBLs, we achieve an entropy of 250! which is already higher than the ultimate ASLR solution that would provide an entropy of 2^{32} on a 32-bit system. Moreover, besides code transformation, we randomize the location of each data section to achieve a fully-randomized memory layout.

Even if the adversary knows that the application suffers from a vulnerability, he cannot launch a ROP or return-into-libc attack, since the location and structure of all BBLs have been randomized. In addition, our approach is secure against *disclosure attacks* where the address of a known function is leaked to the adversary which would normally enable him to revert the entire (affected) code segment. This is due to the fact that all offsets between functions and BBLs have been randomly changed. Even if the permutation and the memory layout of one specific instance is known, the adversary cannot assume that the target device is using this instance, since our diversification is applied for each application run.

2.1 Design of XIFER

The design of *XIFER* is depicted in Figure 2. The workflow is as follows: after the app has been loaded by the OS linker into the memory, we first disassemble the application

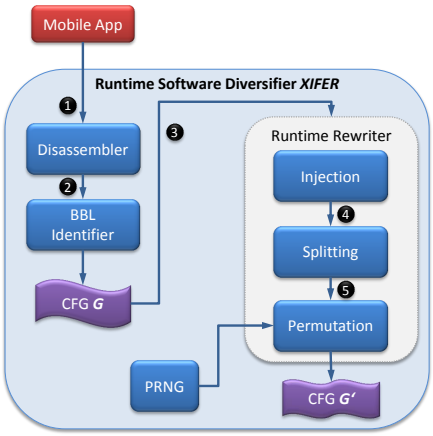

Figure 2: Design of *XIFER*

on-the-fly (step 1). Next, we identify all BBLs that belong to the application and derive the basic control-flow graph (CFG) of the application (step 2). The next steps (steps 3 to 5) involve the diversification of the application which is performed within our runtime rewriter. More precisely, we *inject* new BBLs and nop instructions into the application, *split* BBLs to multiple BBLs, and finally *permute* the BBLs in memory based on the output of a Pseudo Random Number Generator (PRNG). Since we move and inject BBLs in memory, our rewriter needs to adjust all relative memory offsets and branch targets.

2.2 Implementation

The core component of *XIFER* is the binary rewriter which we implemented for the Android/Linux version for Intel x86 and ARM. The code is entirely written in `C++` and consists of 4130 lines of code.

In summary, the main steps involved in the rewriting process of *XIFER* are (1) *loading* the executable, (2) *disassembling* the bytecode on-the-fly, (3) building a *reference graph* of the executable, (4) applying *code transformation*, and (5) finally writing the executable back to memory (*fixation*) so that it can start executing. To accurately rewrite the app, *XIFER* only requires relocation information which is also required to enable conventional ASLR protection.

A particular challenge with regard to BBL permutation is how to handle conditional branches such as the (ARM) `beq 0x8000` (branch on equal) instruction. For this instruction the control-flow might either branch to address `0x8000` or *implicitly* continue at the succeeding instruction. Hence, moving BBLs in memory would break the program's semantics if the program implicitly continues after the conditional branch. To tackle this challenge, we insert unconditional branch instructions at the end of all BBLs that originally end with a conditional branch. This removes all implicit control-flow changes allowing us to randomly split and permute BBLs in memory afterwards.

2.3 Evaluation

We implemented our rewriter and our runtime software diversifier for the popular Linux ELF executable format. To measure the performance overhead we performed microbenchmarks based on a SHA-1 and a BubbleSort algorithm running on a Nexus S device (Android 4). Our results show that *XIFER* efficiently performs runtime software diversity with an average runtime overhead of only 1%. Further, it only induces a load-time overhead of 1s for for disassembling and rewriting a 5MB large executable.

3. CONCLUSION

Our runtime diversifier called *XIFER* accurately mitigates code-reuse attacks by diversifying the structure of an application at *runtime*. At the heart of our tool is our binary rewriter which disassembles application binaries, performs code transformations and assembles new application instances with new memory layouts for each application run, while still covering the entire semantics of the initial program. Currently, we are performing a large-scale evaluation and are extending *XIFER* to support Android native libraries.

4. REFERENCES

[1] F. B. Cohen. Operating system protection through program evolution. *Computer & Security*, 12(6):565–584, 1993.

[2] R. I. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program, 1996. Patent US5559884.

[3] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, 1997.

[4] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *NSPW*, 2010.

[5] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: where'd my gadgets go?

[6] V. Iozzo and R. Weinmann. Pwn2Own contest. `http://dvlabs.tippingpoint.com/blog/2010/02/15/pwn2own-2010`, 2010.

[7] M. Keith. Android 2.0-2.1 Reverse Shell Exploit, 2010. `http://www.exploit-db.com/exploits/15423/`.

[8] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[9] C. Miller and D. Blazakis. Pwn2Own contest. `http://www.ditii.com/2011/03/10/pwn2own-iphone-4-running-ios-4-2-1-successfully-hacked/`, 2011.

[10] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[11] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[12] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[13] A. Sotirov and M. Dowd. Bypassing browser memory protections in Windows Vista. `http://www.phreedom.org/research/bypassing-browser-memory-protections/`, 2008.