Using faults for buffer overflow effects

Pierre-Alain Fouque, Delphine Leresteux, Frédéric Valette

Using Faults for Buffer Overflow Effects

Pierre-Alain Fouque
École Normale Supérieure
45 rue d'Ulm
F-75230 Paris CEDEX 05,
France
pierre-alain.fouque@ens.fr

Delphine Leresteux
Université Paris VII -
DGA Information Superiority
BP7
35998 Rennes Armées,
France
delphine.leresteux
@dga.defense.gouv.fr

Frédéric Valette
DGA Information Superiority
BP7
35998 Rennes Armées,
France
frederic.valette
@dga.defense.gouv.fr

ABSTRACT

Fault attacks have been developed in the cryptographic community to extract secret information on hardware implementations. They have also been used to bypass security checks during authentication processes for example. Here, we show that they can be exploited to make more damage, taking the control of a machine as buffer overflow attacks do for instance. In this short paper, we demonstrate by using one example that countermeasures against buffer overflow must also be used for software running on embedded processors.

Keywords

Buffer Overflow, Fault Analysis, FPGA

1. INTRODUCTION

This paper studies a security vulnerability for embedded processors. Anderson and Kuhn in [1] describe a whole state-of-art on fault attacks. They especially focus attacks on software and list many types of attacks which could affect software implementations like password checking or access rights.

Related Work. Other consequences described for glitch are to turn away unconditional jump to other program segment or an operation on variable into something else. The attacks described in [1] only target systems in order to recover a secret value, to bypass a simple assembly command or to avoid a control test. Here, their goal is to bypass several times a *if* condition to dump the whole memory. Govindavajhala *et al.* in [3] present another fault attack which targets soft memory error in Java virtual machines. Only one bit error allows to take control of a host. As in our attack, the code does not contain any bug. This attack provokes malfunction in address space in *memory*. Two different kinds of pointers point toward the same place in the heap to execute arbitrary code. Contrary to Govindavajhala's article, we produce faults on the program *instructions*. Our faults disrupt element places, while Govindavajhala's fault disrupts element *types*. Even if the results are the same, diverting a program, we target here the registers and not the pointers. The *program counter* register usually seems as a potential target. Corrupting its value allows attacker to jump in another part of code. In state-of-the-art, attacks on *stack pointer* register saved in memory exist, but they only produce wrong branches into program execution. No fault analysis on *stack pointer* during function call has been already published. Here, our attack allows to branch into another valid program. Consequently, we present attack that allows to control the value of the program counter without provoking a fault on it.

Our Results. This paper considers safe implementations that do not use overflowable functions and we target embedded software for which classical countermeasures against buffer overflows are not always provided by the compilers. We analyze the consequences of the faults on the scheduling of program and we obtain similar results as buffer overflow attacks using fault attacks. We have experimented this attack and we show that it can be very efficient.

2. BACKGROUNDS

In this section, we recall buffer overflow and fault characteristics.

Faults. Faults are generated by several external sources: spike on power supply, glitch on the clock, variation on the temperature, laser, electromagnetic radiations ... Each flip flop has its own characteristic which depends on the temperature or on the voltage. Due to a glitch, some flip flops process their inputs before they have to [4]. Data have to be treated too quickly and computations errors occur due to the briefness of the glitch. On the instruction modifications, fault attacks can also be used to avoid the storage of variable in a register, such as the number of rounds for a symmetric cryptographic protocol [1].

Buffer Overflow. Buffer overflow consists in writing data into memory space at different places than the allocated and expected place. This writing can erase useful information on the process. A buffer overflow allows an adversary to take control of a machine by modifying the schedule so that the next executed instruction will not be the expected one. Another instruction located either forward in the program or in another program can be executed instead. If this last one is malicious, the executed code can make damage or can take advantage of the attacked program privileges.

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

3. A NEW KIND OF FAULT ATTACK

After profiling fault effects, we mount our attack. In this attack, we target a specific function whose goal is to verify a password. Then, we explain its success rate and a generalization of this attack that targets other functions, maybe not security function such as control access.

3.1 Attack Context

The attack modifies the value of the *return address* register at the end of a function call as in buffer overflow attack. In our attack, the faults are used to change the *stack pointer* so that it will point towards the input parameters. The attack can be extended to any programs or functions with or without security goals in any context. We have experiment and simulate theoretic and practical results on embedded development board with the microprocessor in using glitch on clock.

3.2 Attack Schedule

This attack uses only one fault to avoid the execution of only one instruction. We illustrate our purpose with a password verification example. Instead of using subfunction like `strcmp`, the developer uses his own function, for illustration `my_strcmp` (i.e. Function A), which realizes a comparison between entered word and the right password, to avoid overflowable function. At each function call, the stack context is saved by several assembly commands. One of this kind of commands consists in an addition (`addi`) to the `stack pointer` (`sp`) with a negative value. We inject a fault during the execution of the `addi` instruction of the `sp` register at the beginning of the function `my_strcmp` in our case study. This precise and located fault avoids the modification of the `sp` value at the beginning of the subfunction, `my_strcmp`. Then we normally go back to the function named `comparison` (i.e. Function B) in our example, as it is shown in Figure 1.

1. We make a fault during the subtraction (`addi sp, sp, -160`) instruction, so the `sp` pointer value is wrong.

2. At the end of the call of the function, `comparison` in our case, the `return address` (`ra`) is retrieved by the load instruction (`ldw ra, 156 (sp)`) (offset from `faulty sp`). In case of our fault, `return address` will point toward input parameters.

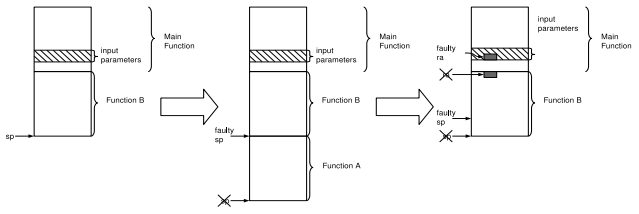

Figure 1: Faulty execution during attack

Input parameters contain address of any function after the password check function. That is why at the end of the function `comparison`, the *return address* register loads the address of the continue function, as if the correct information was entered. Now, the *return address* register contains a faulty value. The final assembly `ret` command ends the fault attack by jumping to execution without stopping due to false password. This scenario needs only one fault and allows to bypass security control. It is illustrated by Figure 1.

4. COUNTERMEASURES

To defeat these attacks, several countermeasures exist and could be implemented. In this section, we present hardware and software countermeasures. While ones prevent from fault attacks, the others detect buffer overflow exploits.

4.1 Countermeasures Against Fault Attacks

Essentially, two different kinds of countermeasures can be used against these attacks. The first category gathers together protections against clock glitch and the second one protections against glitch effects like software redundancy. In first type, hardware components filtering variations can be used. For instance pll for phase locked loop, our glitch attack scenario has been played without results. These techniques are so specific for fault injection means. Other fault injection provokes program diverting.

4.2 Countermeasures Against Buffer Overflow

First countermeasure consists in obfuscating addresses in using randomized address. It prevents opponents from targeting function address after the targeted function. Second countermeasure duplicates *stack pointer* in a variable, in a register, in other part of the stack or in another duplicated stack like Stack Shield [2]. This technique protects from unauthorized modification of *stack pointer*. The last countermeasure that defends from executing malicious code is non-executable stack, for the input parameters.

5. CONCLUSION

This fault attack represents a new real threat for embedded software systems. It is the first time that this association of two different attacks, fault attacks with buffer overflow techniques, is realized. Furthermore, only one fault is needed. It reveals that it is important to adopt the same countermeasures against buffer overflows in software embedded system. The main rule consists of protecting stack memory and registers' integrity during all program execution.

6. REFERENCES

[1] R. J. Anderson and M. G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Security Protocols Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1997.

[2] C. Cowan, S. Beattie, R. Day, P. w. C. Pu, and E. Walthinsen. Protecting Systels from Stack Smashing Attacks with StackGuard. In *Proceedings of the 5th Linux Expo.*, Raleigh, NC, May 1999.

[3] S. Govindavajhala and A. W. Appel. Using Memory Errors to Attack a Virtual Machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

[4] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *WOST'99: Proceedings of the USENIX Workshop on Smartcard Technology*, pages 9–20. USENIX Association, 1999.