JuniKhyat (UGC Care Group I Listed Journal)
ISSN: 2278-4632 Vol-8 Issue-02 Dec 2018

Delving into JTAG Fault Injection Attacks: Strategies and Implications

Sandeep Kumar Mishra
College of Engineering Bhubaneswar

***Abstract*— While fault injection attacks are common in the world of smart cards and microcontrollers, they are still relatively uncommon in the realm of complex embedded microprocessors, such as system on chips found in tablets, smartphones, and automobile systems. The primary reason for this is the difficulty of strategically inserting a flaw to render these attacks on such devices successful. Nonetheless, these devices offer new tools for debugging and development that can be seen as potentially opening the door for attacks. Among these is the JTAG debug tool, which is found on the majority of modern electronics. We introduce the first JTAG-based fault injection attack in this letter. We describe how this works using a privilege escalation attack as an example tool can be used either to check the feasi-bility of this attack by fault injection or to perform an actual attack.**

***Index Terms*—Fault injection attack, JTAG, system on chip (SoC).**

I. Introduction

THE IDEA of using hardware faults to derive sensitive information was first introduced in 1997 [1]. The idea was rapidly generalized and used on protected systems, suchas microcontrollers and smart cards [2]. Many studies on this topic and on how to thwart such attacks are still being con- ducted by researchers to evaluate security weaknesses in the face of such threats [3]. However, with the democratization of smartphones and other connected devices, the manipulation ofsensitive data tends to be done not only by smart cards but alsoby complex and multipurpose system on chips (SoCs). Such devices thus also need to be evaluated against fault injection and combined attacks, but so far, very few intrusive tests havebeen conducted on such systems. The rarity of these tests is mainly due to the difficulty involved in interacting with a tar- geted module integrated in a complex SoC at exactly the right place and the right time. This difficulty is due to their com- plex architectures, their size and the package which coats themon one hands, and to the operating system (OS) running on them, on the other. This letter describes the improper use ofa widespread hardware debugging tool that makes it possible to get round the complexity of the SoCs either to undertakea fault attack or to check weather an attack would be possi-ble using fault injection. The rest of this letter is organized as follows.

Section II describes the contribution of this letter. Section IIIexplains our attack and the example that inspired the useof the debugging tool. Section IV describes the device being tested, the environmental setup and our experiments. Finally, Section V contains the discussion and Section VI our conclusions.

II. Our Contribution

This letter fits in the extended field of “fault attacks” that deals with complex SoC. Our research presents the widespread debugging tool JTAG [4] used as a fault injection vector. Untilnow JTAG was presented in the literature on hardware secu- rity only as a way of dumping and snooping data [5], [6]. However, considering the *debug access* function of the JTAG used by debugger tools that access the internals of a chip, making its resources and functionality available, modifiable and stoppable, highlights the fact that we have at our disposala path for fault injection into a running program.

These debugging capabilities have been already exploitedin [7] to implement a combined attack to load and run a mali- cious code. The authors explain how they can make former software attacks still efficient by modifying the code of func- tions with the JTAG. In this letter, we propose to use the JTAGas a fault injection tool. We describe how the JTAG can be used to set up a disruption when a program is running exactly as in the classical fault injection attacks. Using an exampleon an Android-powered device, we illustrate this new path for fault injection in a JTAG fault injection attack (JTAG-FIA) forprivilege escalation.

III. JTAG-FIA

The JTAG-FIA is based on the JTAG debugging capacities to change on the fly a value linked to the system security with the aim of degrading it.

A. JTAG-FIA Principle

In a program or an OS, a security function f to be degraded is targeted. A set of rules $R = \{r_0, \ldots, r_n\}$ determine the behavior of f. Let f_R be this function. First, the rules are reversed in order to detect which rule $r_i^* \in R$ has a security impact and any comparison operation based on a value v in memory.

Then, with JTAG, the process of f_R is halted before the rule r_i^* is applied. The memory value is changed: $v \rightarrow v^r$, which changes the rule into a new rule: $r_i^* \rightarrow r_i^{*r}$. Finally, $R \rightarrow R^r$

and thus f_{R^r} do not fulfill its intended purpose. When the

```
user@android:/ $ cat /proc/kallsyms
...
0000000000000000 D kptr_restrict
...
0000000000000000 d pfifo_fast_ops
0000000000000000 D noqueue_qdisc_ops
0000000000000000 D noop_qdisc_ops
...
```

Fig. 1. Arranged list of `0`-masked addresses of memory `.data`.

process execution is continued, f no longer guarantees security. In the following section and as an example, this principle is applied in a privilege escalation attack on a function working with the user privilege in the Android OS. *JTAG-FIA Application in Privilege Escalation*

A privilege escalation is the act of exploiting a bug, design flaw or configuration oversight in an OS or software application to increase access to resources that are normally protected from an application or user. The result is that an application with more privileges than intended by the application developer or system administrator can perform unauthorized actions.

1) Principle of Our Attack: Let f_R be a function displaying some critical information. Let $v \in \mathbb{N}$, $r_i^*(v) \in R$ be the rule that determines which privilege is needed to access the information. By definition $r_i^*(0)$: "any user can access," $r_i^*(1)$: "only privileged users can access" and $\forall v \geq 2$, $r_i^*(v)$: "no user can access" where "users" is an entity that can access to these information (human user, program, function, etc.). With the JTAG, the value v stored in memory is targeted and set to 0 in order to display the critical information regardless of any privilege.

2) Detailed Attack: To perfectly understand the attack and anticipate all possible difficulties, some very specific technical details related to Android are required. They are detailed here. In Android OS and more generally in Linux systems, the kernel symbol addresses are displayed by the `/proc/kallsyms` file. However, exposing these pointers provides an easy target for kernel write vulnerabilities, since they reveal the locations of writable structures containing easily triggerable function pointers. Also for the purpose of security, the function `s_show` (f_R) in the `kallsyms` file use the `%pK` format specifier (r_i^*) which is designed to hide exposed kernel pointers, specifically via `/proc` file interfaces. The behavior of `%pK` depends on the integer value of `kptr_restrict` sysctl (v). By definition in source code of the Linux kernel, if the `kptr_restrict` value is equal to `0`, there are no restrictions. If `kptr_restrict` is set to `1`, the kernel pointers using `%pK` will be replaced by `0`s unless the current user has privileged permission. Otherwise, kernel pointers printed using `%pK` are printed as `0`s regardless of privileges. Obviously, most of the Android devices provided to customers are in simple user mode and `kprt_restrict` is set to `1`. This value can only be changed by a privileged user. So, the principle of our attack is the following: by a fault injection, force the `kptr_restrict` value to be set to `0`. Making these addresses available for a simple user is a privilege escalation.

IV. Experimentation

A. Experimental Set-Up

1) *Device Under Test:* In our experiment, we used a development board with a widespread octa-core 64-bit ARM Cortex-A53 application processor embedded executing Android OS version 6.0.1—Marshmallow. For our experiment, the board was deliberately left in user mode with no privileges. To communicate with the Android-powered device, we used Android debug bridge (ADB), which is a command line tool without *root privileges.*

2) *JTAG Hardware Debugger:* The debug tool used for the manipulations is a Lauterbach probe. The Trace32 soft of this probe provides a very detailed and tailored GUI that gives an overview of the device and can perform a wide range of operations on it. Moreover it makes it possible to include and automate JTAG routines in a C program as required in our next experiment.

B. JTAG-FIA for Privilege Escalation

The success of the JTAG-FIA depends on its ability to mod-ify the value of `kptr_restrict` which manages access tothe kernel symbol addresses. The main difficulty is findingthe address of the value in memory. In this case, the com- plexity is due to the size of the main memory (RAM) and toits management by the OS running on the SoC. In any case, OSs loaded into the RAM can be seen as hexadecimal values of compiled code and these values depend on the compiler used and the options selected among other things. So to find the `kptr_restrict` address value, the trick is to search for a typical fixed pattern in the memory nearby. Among the huge amount of hexadecimal values, the most easily identifi- able are the ASCII strings of characters. So, the JTAG-FIAfor privilege escalation can be split into two main steps.

1) With the JTAG tool, find the address of a typical fixed ASCII string near the `kptr_restrict` address.
2) From this address, go to the `kptr_restrict` address by modifying the values in memory (i.e., injecting a fault) with the JTAG probe and check if the addresses are displayed with the ADB tool.

The following sections describe how some of the difficulties encountered in these two points were resolved.

1) Find the Address of Typical Fixed ASCII String: Actually, the value of `kptr_restrict` is a 32-bit word inthe `.data` memory section that is either `0x1` or `0x2` whena simple user cannot read the kernel symbol addresses. Ona modern 64-bit architecture with 2^{64} 1 addresses, directly finding which `0x1` or `0x2` match `kptr_restrict` in thememory is a laborious and time consuming task. For example,in our experiment, among the only 2.1GB of data dumped by JTAG means there are still 1 733 598 of `0x1` and 1 349 512of `0x2`. Checking the effect of the change of each of these values is too-long considering our procedure. Fortunately

"String"	Hexadecimal ASCII value
"pfifo_fast"	66696670 61665F6F 00007473
"noqueue"	75716F6E 00657565
"noop"	706F6F6E
"bfifo"	66696662 0000006F
"pfifo_head_drop"	66696670 65685F6F 645F6461 00706F72
"pfifo"	66696670 0000006F

Fig. 2. Little endian hexadecimal values targeted in the memory.

the memory .data field of Linux kernel contains ASCIIstrings that can be considered as markers. As described in several Linux kernels files [8], for example, those locatedin the /net/sched/ directory, part of these strings arethe .id values of structures. In particular, the ones in the sch_generic.c and sch_fifo.c files, which are ASCII strings stored just after the address of the kptr_restrict value (see Fig. 1). However, dumping and parsing the whole system memory to find character strings is time consuming, especially with a set of 64-bits addresses. This is why a trickis used to find the approximate address to begin the search.On all Android devices, and for all unprivileged users, the version of the kernel and the compiler used to build it can be identified by sending the command line "adb shell "cat /proc/version"" through the ADB tool. In our experi- ment, the Linux kernel version is 4.1.15 and the compiler is gcc version 4.9.x-Google. This makes it possible to download the same kernel version on [8] and the same com-piler on [9], to build it, and to obtain an approximate memory map. When the kernel is built, the System.map file pro- vides the same information as the "/proc/kallsyms" file with no 0-masked values but with some potential offsets inthe addresses due to the compilation options. Nevertheless,the address of the _text symbol defines reserving memoryblock for the kernel _text and _data section. These manip- ulations reveal the address to start dumping: $@_{SD}$ (in our case $@_{SD}$ 0xFFFFFFC000000000).

The first step is quite simple: dump the kernel by small packages, one by one, starting at the previously defined address and then parse these packages to find the required ASCII strings. Considering the .id values included in the sch_generic.c and sch_fifo.c files, the ASCII val- ues searched for are the ones detailed in Fig. 2. Using thesame principle as pattern matching algorithms, in approxi- mately 52 s, 27 packages of 78 KB are dumped and parsed until the required values are found in the same order as in the /proc/kallsysms file. Fig. 3 summarizes this step and shows several addresses, where the ASCII strings of charac- ters are found. Only the "noop" string appears several times before all the strings are found in the right order. Fig. 4 shows amemory dump with the JTAG probe at the addresses, where thestrings have been identified. The address of "pfifo_fast"in the memory is the starting point of the next step of our JTAG-FIA for privilege escalation.

"String"	Identified at the address:
"pfifo_fast"	0xFFFFFFC000D6CD88
"noqueue"	0xFFFFFFC000D6CE10
"noop"	0xFFFFFFC0009E76B0 0xFFFFFFC000A15ECC 0xFFFFFFC000C0E2F0 0xFFFFFFC000D6CE98
"bfifo"	0xFFFFFFC000D6D030
"pfifo_head_drop"	0xFFFFFFC000D6D0B8
"pfifo"	0xFFFFFFC000D6D140

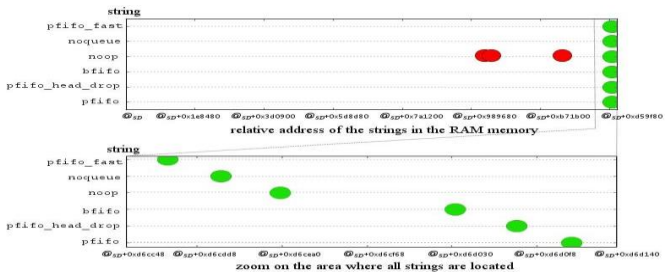

Fig. 3. Upper part shows the relative positions of the addresses of theASCII strings into the RAM memory. The lower chart, gives the ASCIIvalue addresses of each string found in the memory from the address $@_{SD}$ =0xFFFFFFC000000000.

2) Go to the kptr_restrict Address: The laststep is a straightforward scan from the start addressdefined in the previous stage up to the kptr_restrictvalue. The routine is the following: (init) *start_addr* 0xFFFFFFC000D6CD88 and *addr* *start_addr*. 1) Read the32-bit word value at the address *addr* *start_addr* - 0x4, ifthe latter is equal to 0x00000001 or 0x00000002 change it to 0x00000000. Then, from the ADB interface, = sendthe command line to check if the kernel symbols are avail-able: adb shell cat /proc/kallsysms. If the valuesreturned are 0-masked, then re-establish the original valueof the 32-bit word and read the next word, i.e., repeat step1). Otherwise, the attack succeeded and the kernel symbolsare available for simple users (end). During the execution ofthis routine, only six 0x1 and one 0x2 were counted. Thekptr_restrict is the last 0x2 value found at the address 0xFFFFFFC000D6B320.

V. Discussion

In this letter, we describe how JTAG can be used to modify the code on the fly in order to tamper with any OS. We didnot crash the system but simply hijacked it. The condition for the success of the attack is accessing the JTAG interface. For the sake of practicality, most development boards have their JTAG port available and unlocked. But what about commercialsystems? Two approaches are possible.

A. Try to Access the JTAG Interface

Nowadays many SoC manufacturers are aware of the risks associated with JTAG and consequently provide secu- rity solutions. As explained in some academic articles, such

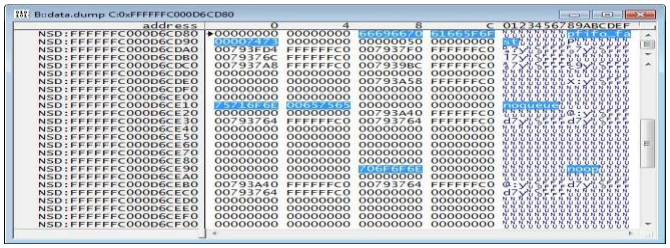

Fig. 4. ASCII in the `.data` field of the memory kernel in the RAM memory displayed by JTAG debugger.

as [5] and [10] it is possible to take preventive measures to ensure JTAG integrity or to directly secure the scan chainsas in [11] and [12]. The detection of a JTAG misuse is also possible as explained in [13]. However, many manufacturers make do with lock access mechanisms as in [14].

It is therefore interesting to study these protection mecha- nisms. As shown in [15], it is possible to imagine new attack paths and several fault injections in order to bypass such security mechanisms and access JTAG.

B. Use the JTAG to Check the Feasibility and the Effect of Fault Injection

In the introduction, we explained why it is so complex toset up a fault injection attack on SoCs. Even if the JTAG is disabled on the device an attacker would like to attack, it couldstill be used on a development board similar to this device to simulate whether an attack is feasible. Our privilege escalationattack showed that only one bit has to be flipped. By using an other physical quantity, for example, electromagnetic field, it ispossible to measure the emission field related to the call of thefunction and detect it. This kind of detection is already used for encryption on SoC [16]. Once the instant of the function call is detected, an electromagnetic pulse is injected to disrupt it, as reported in [17]. The setup of this kind of manipulationis time consuming but by repeating the call to the func- tion and the EM pulse injection it is statistically possible to succeed.

Both approaches show that the JTAG can also be seen as a support for setting up a fault attack.

VI. Conclusion

Using the JTAG debugger, we have shown how to alter the code dynamically and meddle with any form of

of OS. We can change a value associated with system security by utilizing solely the JTAG's debugging capabilities. JTAG is a novel fault injection tool that can be used to carry out or mimic the viability of an attack using fault injection directed at a sophisticated embedded microprocessor. While the JTAG-FIA example presented here is optimized for Android-powered smartphones, the idea is applicable to any OS running on devices that have a JTAG access port. Our method allows one to assess the viability or launch an attack straight away.

References

[1] D. Boneh, R. A. DeMillo, and R. J. Lipton, “On the importance of check- ing cryptographic protocols for faults,” in *Advances in Cryptology—EUROCRYPT’97* (Lecture Notes in Computer Science). Heidelberg,Germany: Springer, 1997.

[2] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, “The sorcerer’s apprentice guide to fault attacks,” *Proc. IEEE*, vol. 94, no. 2, pp. 370–382, Feb. 2006.

[3] C. Giraud and H. Thiebeauld, *A Survey on Fault Attacks*. Boston, MA, USA: Springer, 2004, pp. 159–176, doi: 10.1007/1-4020-8147-2_11.

[4] *IEEE Standard for Test Access Port and Boundary-Scan Architecture— IEEE Computer Society*, IEEE Standard 1149.1-2013, May 2013.

[5] K. Rosenfeld and R. Karri, “Attacks and defenses for JTAG,” *IEEE Des.Test. Comput.*, vol. 27, no. 1, pp. 36–47, Jan./Feb. 2010.

[6] S. Willassen, “Forensic analysis of mobile phone internal memory,” in *Proc. Adv. Digit. Forensics IFIP Int. Conf. Digit. Forensics Nat. Center Forensic Sci.*, Orlando, FL, USA, Feb. 2005, pp. 191–204.

[7] F. Majeric, B. Gonzalvo, and L. Bossuet, “JTAG combined attack— Another approach for fault injection,” in *Proc. 8th IFIP Int. Conf. New Technol. Mobility Security (NTMS)*, Larnaca, Cyprus, Nov. 2016, pp. 1–5.

[8] *The Linux Kernel Archives*. Accessed: Mar. 2016. [Online]. Available: https://www.kernel.org

[9] *Native Development Kit*. Accessed: Jun. 2016. [Online]. Available: https://developer.android.com/ndk

[10] J. Backer, D. Hely, and R. Karri, “Secure and flexible trace-based debug-ging of systems-on-chip,” *ACM Trans. Design. Autom. Electron. Syst.*, vol. 22, no. 2, pp. 1–25, Dec. 2016, doi: 10.1145/2994601.

[11] L. Pierce and S. Tragoudas, “Multi-level secure JTAG architecture,” in *Proc. IEEE 17th Int. On-Line Test. Symp.*, Athens, Greece, Jul. 2011,pp. 208–209.

[12] J. Backer, D. Hély, and R. Karri, “Secure design-for-debug for systems- on-chip,” in *Proc. IEEE Int. Test Conf. (ITC)*, Anaheim, CA, USA,Oct. 2015, pp. 1–8.

[13] X. Ren, V. G. Tavares, and R. D. S. Blanton, “Detection of illegiti- mate access to JTAG via statistical learning in chip,” in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Grenoble, France, Mar. 2015,pp. 109–114.

[14] *Configuring Secure JTAG for the i.MX 6 Series Family of Applications Processors—Application Note*, document An4686 rev. 1, NXP, Eindhoven, The Netherlands, Mar. 2015. [Online]. Available:http://www.nxp.com/files/32bit/doc/app_note/AN4686.pdf

[15] S. Skorobogatov and C. Woods, “Breakthrough silicon scanning discovers backdoor in military chip,” in *Cryptographic Hardware and Embedded Systems—CHES 2012* (LNCS 7428), E. Prouff and P. Schaumont, Eds. Heidelberg, Germany: Springer, 2012.

[16] J. Longo, E. De Mulder, D. Page, and M. Tunstall, “SoC it to EM: Electromagnetic side-channel attacks on a complex system-on-chip,” in *Proc. 17th Int. Workshop Cryptograph. Hardw. Embedded Syst. (CHES)*,St.-Malo, France, Sep. 2015, pp. 620–640.

[17] S. Ordas, L. Guillaume-Sage, and P. Maurine, “EM injection: Fault model and locality,” in *Proc. Workshop Fault DiagnosisTolerance Cryptograph. (FDTC)*, St.-Malo, France, Sep. 2015, pp. 3–13,doi: 10.1109/FDTC.2015.9.