Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0 -

Guillaume Barbu[1,2] and Hugues Thiebeauld[3,⋆]

[1] Oberthur Technologies, Innovation Group,
Parc Scientifique Unitec 1 - Porte 2,
4 allée du Doyen George Brus, 33600 Pessac, France
[2] Institut Télécom / Télécom ParisTech, CNRS LTCI,
Département COMELEC,
46 rue Barrault, 75634 Paris Cedex 13, France
[3] RFI Global Services Ltd,
Pavilion A, Ashwood Park, Ashwood Way,
Basingstoke, Hampshire, RG23 8BG, United Kingdom

Abstract. Up to now devices in charge of performing secure transactions mainly remained limited regarding their functionalities. However the trend has recently gone towards an increasing integration of features and technologies, which could potentially represent a source of additional threats. This article introduces an innovative attack exploiting advanced functionalities and offering unrivalled opportunities. This attack targets specifically the multithreaded systems featuring network capabilities. By the way of a network flooding we show how a process can be interrupted at the precise time a sensitive operation is being executed. This interruption aims at subsequently modifying the execution context and consequently breaking the sensitive operation. The practical feasibility of this attack is illustrated on a Java Card 3.0 Connected Edition platform. This description reveals that going through with the full attack scenario is not obvious. However this apparent complexity must not conceal the potential breach, which may significantly alter any application running on the system. Finally the goal of this work is to emphasize that the increasing products complexity may generate new security issues rather than to highlight a specific weakness on released products.

Keywords: Fault Injection, Logical Attack, Multithreading, Network Flooding, Java Card 3, Technological Convergence.

1 Introduction

Every software developer knows the usefulness of debug sessions in an application development process. To be in a position to debugging, the developer needs to use some specific tools enabling him to set breakpoints in the middle of a code execution. He has then the ability to control the internal process flow by

⋆ Part of this work done while at Oberthur Technologies.

E. Prouff (Ed.): CARDIS 2011, LNCS 7079, pp. 18–33, 2011.
© IFIP International Federation for Information Processing 2011

modifying the execution context. Such a capability on released device would represent an outstanding breach. Indeed, it would allow to alter any application running on the defeated system.

This article introduces a new attack revealing a malicious mean to set a breakpoint. This attack targets multithreaded systems with network capabilities. By exploiting a network flooding we demonstrate how a multithreaded system can be abused to artificially freeze an application during a sensitive operation process. Combined with a temporary illegal access to the execution context in memory, we describe how the context modification may alter the security of the application when its process is resumed.

As a proof of concept we detail a practical attack on a Java Card. The recently released Java Card 3.0 Connected Edition specifications [1, 2, 18, 3] associate the multithreading and network connectivity features with the Java Card technology. To fit with the particular context of secured Java Cards, our description reveals that some inherent system protections must be circumvented requiring a fault injection [4, 5, 6, 7]. In spite of its apparent complexity we have successfully implemented the attack to alter the security of a Java Card web application. We also present different ways to withstand such attacks.

Beyond the practicability of the attack this work seeks to highlight that any feature is a potential source of attack, even though the links with security are not obvious. The article is subsequently organized as follows : In Section 2, we briefly present the involved mechanisms. Then we expose the generic attack concept in Section 3 and detail in Section 4 a complete attack path on a JC3.0. Finally, in Section 5 we discuss the protections preventing this attack and we use this example to emphasize the importance of the implementation to fit a high level of security.

2 Involved Mechanisms

As stated above, a prerequisite for our attack concept is the support of multithreading and network communications over standard protocols. This section intends to outline these mechanisms to set the basement of our work.

2.1 Multithreading

Our concept is grounded on the multithreading capacity of the targeted system, which allows the concurrent execution of different processes. In this work, we only consider multithreading on single-core devices. To process several threads simultaneously, the system assigns resources to a thread for a given time slice before switching to another. The entity in charge of distributing resources to the threads is the scheduler. Numerous rules can be used to decide when the scheduler will order a thread switching, *i.e.* to set the size of the so-called time slice. For instance thread switching can be triggered by a timer, an instruction counter, control flow breaks, access to certain resources, *etc.*

When a thread has consumed its allocated time slice, the scheduler orders a switch. This switch should be processed as follows :

- The current thread's execution context is saved.
- The next thread is elected and its execution context is loaded.[1]

The different threads are then successively given access to the system resources.

The attack concept does not directly target the multithreading but rather lies on an abuse of this feature. Next section introduces the feature we take advantage of to achieve this abuse.

2.2 I/O Network Interfaces

We consider in the scope of this work a device providing a logical network interface supporting standard network protocols (TCP, HTTP(S), ...) over physical I/O interfaces. The aim of this section is to set the system architecture assumed in the remainder of this work.

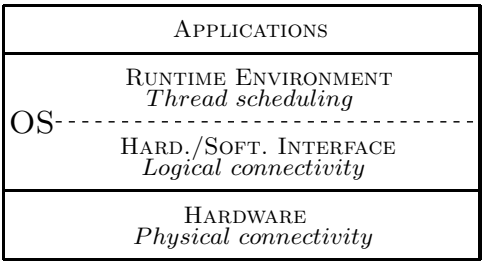

Fig. 1. Alleged architecture of the system

As depicted in Fig. 1, these interfaces are most likely not part of the so-called runtime environment (RE), but belong to lower layers. According to this statement, we can expect that some incoming requests are handled in these lower layers only and do not reach the RE. Therefore they do not enter the system's multithreading mechanism.

This last statement is a key element rendering our attack concept practicable, regardless of the targeted system. For the sake of clarity, Fig. 2 illustrates it with different requests. As depicted on the figure, requests $\mathcal{B}$ and $\mathcal{D}$ handling leads to a thread creation within the RE, whereas requests $\mathcal{A}$ and $\mathcal{C}$ are handled by the system before entering the RE.

Now we have set the basement of our attack concept, introducing the required mechanisms and properties of the targeted system, the attack concept itself can be exposed.

[1] The next thread election can possibly take into account the concept of thread priority. This concept will not be further considered in our context since an attacker able to start new threads should also have the ability to modify their priority.

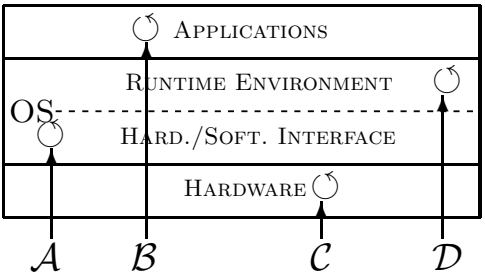

Fig. 2. Different requests handling

3 The Attack Concept

This section introduces the attack principle in a generic way. The goal is to emphasize that this threat may potentially affect a wide range of systems. On the other hand we will see that the attack success is closely related to implementation choices in the platform, providing some leads to find adequate protections later on.

The attack aims at altering a sensitive execution flow at a precise time. To achieve this, two steps must be performed as follows :

- Freezing the application execution at time T_0. It comes to setting a breakpoint on a specific operation.
- Altering the execution context available in memory to change the application behaviour when it is resumed.

In the scope of this article, we assume that the thread scheduler is based upon a timer. A certain amount of time is then allocated to each thread. When an execution exceeds the time slice T, the scheduler stops the process and switches to the next thread in the queue. This hypothesis is obviously not the single way to implement multithreading. However for the sake of clarity, we intentionally focus our description on one kind of scheduler. The adaptation of this attack to other schedulers may be subject to future works.

Our attack relies on the corruption of the multithreading system to force the interruption of an application. The objective is to cheat the scheduler, so that the thread switch occurs at T_0 rather than T. This is obtained by sending a sequence of requests to the device when the targeted thread is being processed. As the process in charge of the network requests is unlikely to be handled as a thread, the time of processing initially devoted to the current thread is lost. As a consequence the thread execution is curtailed, as depicted in Fig. 3.

T_0 is then adjustable depending on the number of network requests sent. To appropriately determine T_0, it is better to have an idea of the thread execution flow. Nevertheless, the code knowledge may not be necessary as side channel analysis may provide sufficient information, depending on the attacked system.

Once the targeted thread is frozen, the scheduler switches to the next ones in the queue. During this time the context of the sensitive code is available in

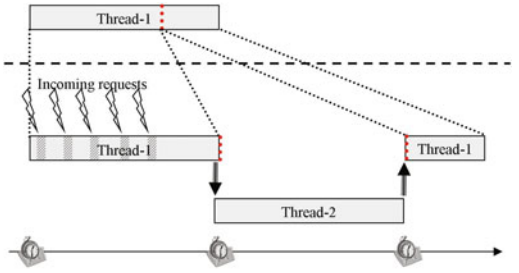

Fig. 3. Normal (up) and curtailed (down) execution of a thread

memory. An attacker in position of executing a malicious application would have then the opportunity to get to the context in memory and alter it. Depending on the attacked system, the right to load and run a application can be more or less restrictive. However in the context of a multi applicative platform these rights necessarily exist, indeed can be forced.

The remaining issue to achieve this attack is memory access. In some open systems the volatile memory remains fully available. But some systems isolate the memory access to respective areas. Therefore this restriction does not allow a thread to get to the execution contexts. To successfully perform the attack, the isolation mechanism must be overcome. The next section illustrates how such a protection has been circumvented on a Java Card by the mean of a physical perturbation.

Once memory access is obtained, a full range of possibility is offered to the attacker. The control of the execution context of the thread gives access to its program counter, local variables and execution stack. Although access to these data obviously stands as a compromission of the system, an attacker has no guarantee that she will be able to take benefit of it. On the other hand, provided she has properly adjusted T_0 , well-chosen alterations would break almost any security operation. The complete attack scenario is depicted on Fig. 4.

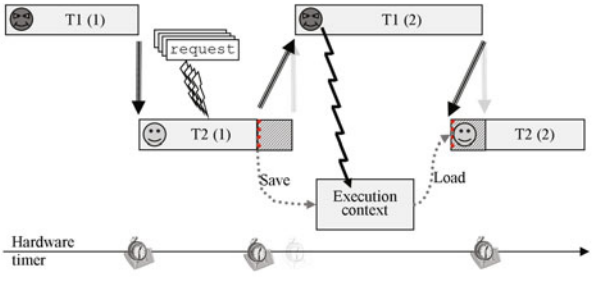

Fig. 4. The complete attack scenario

The attack potentially concentrates several issues which strongly depends on the kind of attacked system. But its consequences may be tragic for an application, even if the code has been proficiently secured. Furthermore this attack also underlines that the security of an application has a value only if the platform underneath is secured enough.

4 Practical Implementation on a Java Card

This section details the full attack scenario we have put into practice on a recent device to illustrate the feasibility of this concept and outline its consequences.

4.1 Context of the Attack

The Attacked Platform. With regards to the different features involved in the attack, a device implementing the JC3.0 specifications appears to be a potential target. Indeed, it is a security device offering both multithreading and network communication support. Furthermore, such platform may allow post-issuance application loading, as long as the application is well-formed.

The Target Application. We consider in the remainder of this section an application $\mathcal{T}$ offering sensitive services. Access to those services requires an authentication, achieved through a signature. $\mathcal{T}$ then contains the following lines of code (or equivalent) :

```
1. if (sig.verify(inBuf,inOf,inLen,sigBuf,sigOf,sigLen) != true)
2.   accessDenied();
3. else
4.   accessGranted();
```

The bytecode sequence that is actually executed on-card is then :

```
0E. invokevirtual #4    <javacard/security/Signature.verify>
11. iconst_1
12. if_icmpne 0x1C (+10)
15. aload_0             <app/Target this>
16. invokespecial #5    <app/Target.accessDenied>
19. goto 0x20 (+7)
1C. aload_0             <app/Target this>
1D. invokespecial #6    <app/Target.accessGranted>
```

Attack Goal. The attack aims at gaining access to the sensitive services without producing a valid signature.

4.2 The Attack Concept Key Assumptions

We have stated in the previous section that the success of our attack concept relies on a couple assumptions validity. This section details the validation of these assumptions on the attacked platform.

Loading the Attack Application. Loading application on the platform is not an obvious right for Java Card users. This capacity is generally limited by the knowledge of authentication keys through GlobalPlatform [8]. However, we consider that an attacker may be able to load an application. This ability can have various origins :

- The load keys are known (this knowledge being either legitimate or not).
- One or several fault injection(s) can lead to a breach in the GP implementation on the card.

Loading and executing a malicious application $\mathcal{A}$ have two rationales. First it should permit the modification of the targeted thread's execution context. But it is also in charge of ensuring the expected thread scheduling scenario. Details of its implementation are given along the attack path.

How to Access and Corrupt the Java Frame. The first challenge is to access the memory and to identify the execution context. Considering a JC3.0, we are interested in the following elements (refer to [9] for a detailed description of Java *Threads* and *Runtime Areas*) :

- the Java program counter : the address of the currently executing instruction in the current method of the current frame.
- the stack of frames : a frame is pushed onto the stack when a method is invoked and is popped out when this method completes.
- in the frames : the local variables and the operand stack.

We intend to reach these values by forging a fake byte array. For that matter, we consider a type confusion provoked by means of a fault injection.

The fault attack. At CARDIS'10, Barbu et al. proposed a combined attack provoking a type confusion and permitting to forge an object's reference and content [10]. In our context, this attack turns out to have a couple of advantages :

- A single physical attack of the device is required, a perturbation during the execution of a `checkcast` instruction for instance.
- Since forged references are persistent :
  - The fault injection can be the first step of our attack scenario.
  - Once one perturbation has been successful, a failure in the following steps will not require to start again from scratch.

We successfully applied this technique to the attacked platform. The physical perturbation was achieved using a laser. Its success depends on a couple of parameters (time, location and wavelength of the beam) found experimentally.

Accessing the Java frame. We assume that execution contexts are saved in volatile memory on thread switching. The type confusion is used to forge a byte array in memory in order to access the execution context of $\mathcal{T}$. We also assume the internal representation of a Java array contains a pointer (say a 32-bit word) to its content in memory, as exposed in [11] and depicted within Fig. 5.

To build a fake array, we only have then to set our "confused object" fields to appropriate values. Then, we expect to be able to access the memory as if it was the content of the forged array. This process is illustrated in Fig. 5.[2]

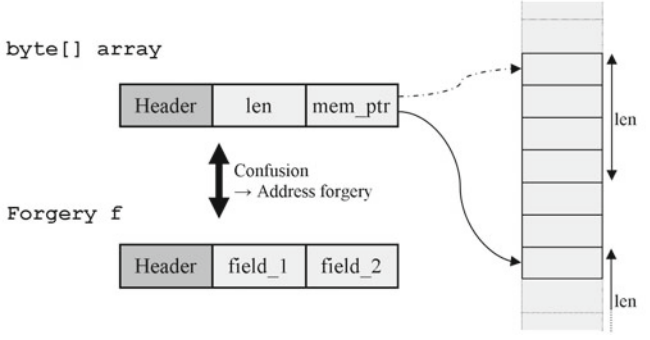

Fig. 5. Confusion between instance of two classes in order to forge an array's address

Once we have gained access to memory, we need to identify the frame within the forged array and to figure out its internal structure.

Finding and learning the structure of the Java frame. To locate the frame, we can take advantage of a straightforward linear memory allocation mechanism. According to the scheduling scenario of our attack concept, initializing a new array with obvious values when $\mathcal{A}$ is resumed permits to delimit the memory used by the targeted application. Furthermore, we have run a training session of our attack in order to learn the structure of frames on the platform. For that matter, we have built a target application that interrupts itself with easy-to-detect `short` values in local variables (`0x1903` for instance) and on the operand stack (`0x1902` for instance). The dump array obtained when $\mathcal{A}$ is resumed is depicted in Fig. 6.

We can then detect the frame and gain sufficient information on its structure :

- <number of local variables : nb_loc> <nb_loc * local variables>
- <maximum stack size : max_stk> <max_stk * operand values>
- <current top of stack>
- <jpc>

How Ping Flooding Affects Application Execution. We have stated in Section 2.2 that some incoming requests do not require the attention of the JCRE. An Internet Control Message Protocol (ICMP) *echo* request (a ping) is a typical example of such a request. Our claim is that when a ping request is incoming, the processor handles it whereas in the meantime the scheduler's timer is still running. We can then manage to shorten a thread's execution as

[2] See Appendix A.1 for implementation details.

```
0x0000 : 55 55 55 55  55 55 XX XX  XX XX XX XX  XX XX XX XX
                        <proprietary data>
0x0030 : XX 14 00 01  00 BA E2 01  00 E2 04 01  00 F2 45 00
0x0040 : 00 00 70 00  00 03 19 00  00 03 19 00  00 03 19 00
0x0050 : 00 03 19 00  00 03 19 00  00 03 19 00  00 03 19 00
0x0060 : 00 03 19 12  E1 53 12 00  12 E1 08 00  00 00 02 19
0x0070 : 00 00 02 19  00 00 02 19  00 00 02 19  00 00 02 19
0x0080 : 00 00 02 19  00 00 02 19  00 00 02 19  00 00 20 00
0x0090 : 00 00 49 00  XX XX XX XX  XX XX XX XX  XX XX XX XX
                        <proprietary data>
0x00D0 : XX XX XX XX  XX XX XX XX  XX 55 55 55  55 55 55 55
0x00E0 : 55 55 55 55  55 55 55 55  55 55 55
```

Fig. 6. Memory dump from the forged array

we please, the number of instructions actually executed within a time slice being reduced. To validate this claim, we have run a thread incrementing a counter on the attacked platform. Fig. 7 presents the value reached by the counter after a given amount of time against the number of pings sent in the same time. This proves that the number of instruction executed within the thread is reduced when the system is flooded with pings, since the value reached by the counter is representative of the number of instructions executed.[3]

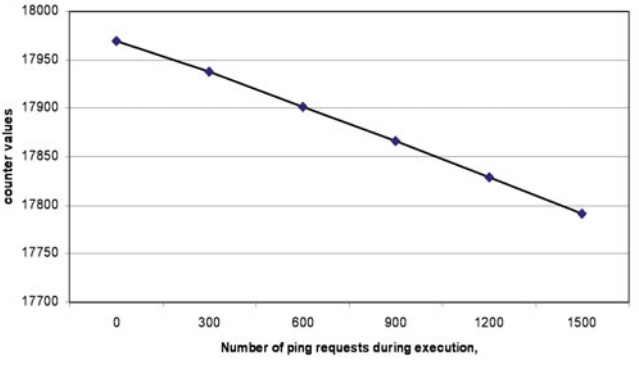

Fig. 7. Influence of communication on instructions execution

This technique could be assimilated to a well-known attack in the network security field : *ping flooding* [12,13]. Ping flooding usually aims at consuming the bandwidth of the targeted system in order to provoke a *Denial of Service* (DoS). Our approach is different as the aim is here to consume the time allocated to the targeted thread in order to curtail it.

[3] Implementation details are given in Appendix A.2.

4.3 The Practical Attack

The attack is divided into three steps detailed within this section.

- In the first step, we achieve the preliminary work to ensure both the access to $\mathcal{T}$'s frame and the scheduling scenario;
- In the second step, we force the "breakpoint hitting" with I/O flooding;
- In the third step, we use the fake array to corrupt $\mathcal{T}$'s frame.

Preliminaries. With regards to the global illustration of the attack concept, this step corresponds to the first segment of the "evil" thread's execution (T1(1) in Fig. 4). The aim of this step is to procure a way to access the memory where the execution context of $\mathcal{T}$ will be stored. This is achieved as presented in Section 4.2.

To ensure the predicted thread scheduling scenario, the application only has to start a new thread, and force its interruption for a certain amount of time (via the `Thread.sleep()` method). Within that time, $\mathcal{T}$ is launched in a new thread. On the next thread switching, $\mathcal{A}$'s thread will then become active again. This last statement implicitly assumes that no other thread is concurrently running on the platform, or at least that the attacker's thread will be the next one to be executed. We can then focus on $\mathcal{T}$'s execution and when to force its interruption.

Setting the Breakpoint. The aim of this step is to force a thread switching at a precise point during $\mathcal{T}$'s execution. It corresponds to the first segment of the targeted thread's execution of the attack concept illustration (T2(1) in Fig. 4). The challenge at this step is to "synchronize" the pings and the thread's execution. Working on a smartcard, the power consumption analysis can again reveal a strong ally at this step. Actually, we can monitor bytecode instruction execution through the power consumption of the card (as stated in [14]). Therefore, the exact knowledge of the code does not appear necessary to achieve the attack.

The attacked platform comes within a USB smart card connector and communicates as an Ethernet Emulation Model (EEM) device according to the specification [15]. The first task is then to adapt our power consumption acquisition module to monitor power consumption behind the USB smart card connector where the Java Card is plugged (*cf.* Fig. 8).

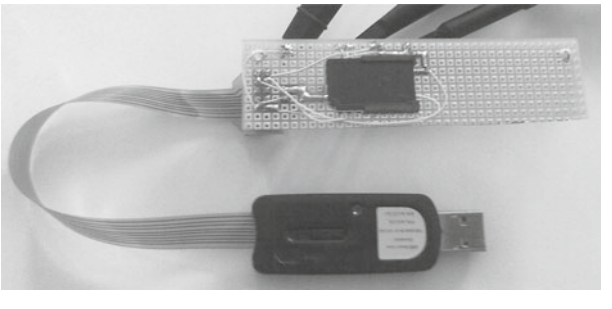

Fig. 8. USB smartcard acquisition module

We can then achieve and monitor the ping flooding of $\mathcal{T}$.

Fig. 9 shows the power traces of $\mathcal{T}$'s execution. On the first power trace, the signature verification is easily identified. The following traces depicts the same execution with an increasing number of ping requests (the numerous peaks on the traces). As we can see, the cryptographic operation is executed more or less shifted depending on the number of pings received during the thread's execution.

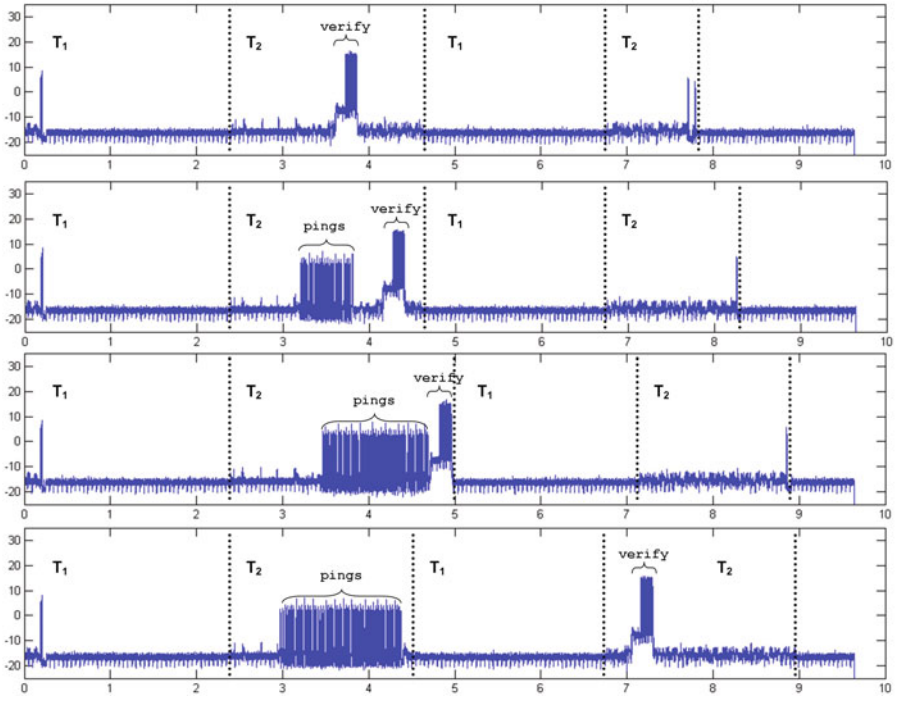

Fig. 9. Execution of the two threads with various number of ping requests (respectively 0, 10, 30 and 40). T_1 and T_2 refer respectively to the attacker's and the target thread.

Based on experimentations, an average sequence of 37 ping requests during the execution of $\mathcal{T}$ causes its interruption after the `verify` method returns but before the execution of the conditional branching. This corresponds to the third power trace in the figure. Actually other "breakpoints" may also allow an attack.

The previous section has given us a mean to read/write the volatile memory. In this section we have exposed how we manage to set the so-called breakpoint within the attacked application $\mathcal{T}$. To complete the attack, we will now try to modify $\mathcal{T}$'s frame in order to bypass its security.

Corruption of the Java Frame. From application $\mathcal{A}$, we can now corrupt $\mathcal{T}$'s frame. We are then literally spoilt for choice in order to bypass the application's security :

- Set the jpc to a given value in order to modify the execution flow,
- Assign given values to references or integral values in the operand stack to have a method executed on the wrong object or with wrong parameters, or return a wrong value,
- Assign given values to references or integral values in the set of local variables, with the same consequences.

With regards to the current state of the art of fault injection, these possibilities are quite outstanding. In a manner of speaking, completing the three previous steps enhances tremendously the initial fault model. We present hereafter one of the numerous way to render the security check of $\mathcal{T}$ useless by modifying the value of the Java Program Counter.

Modification of the jpc. As expected, the signature verification has failed and the conditional branching at line 0x12 leads the execution flow in the "`accessDenied` branch". Because of the flooding, the so-called breakpoint is "hit" at line `0x1D` (`invokespecial #6 <accessDenied>`).

$\mathcal{A}$ is then resumed and we can read $\mathcal{T}$'s frame and identify the jpc value (Fig. 10).

```
0x0040 : XX XX XX XX  XX XX XX XX  XX XX 08 00  01 00 38 FC
0x0050 : 01 00 78 F2  01 00 D2 FE  00 00 05 00  00 00 00 02
0x0060 : 01 00 46 F2  00 00 00 00  00 00 01 00  E7 D2 66 00
0x0070 : 0B D4 07 00  01 00 38 FC  00 00 01 00  00 00 05 00
0x0080 : 00 00 00 02  01 00 46 F2  00 00 00 00  00 00 01 00
0x0090 : 00 00 18 00  00 00 1D 00  XX XX XX XX  XX XX XX XX
```

Fig. 10. Memory dump from the forged array

An easy way to overcome the signature invalidity is then to modify the jpc in order to jump "manually" in the desired branch. That is to say to say to modify the jpc value from `0x1D` to `0x16`. This is done by a mere affectation in the forged array : `ac.array [jpc_offset] = 0x16;`

When the scheduler switches back to $\mathcal{T}$, its execution continues according to the frame. That is to say at the offset we have just set. The next executed instruction will then be the invoke of the `accessGranted` method. As a consequence, we gain access to the privileged method, although we do not have the private key to produce a valid signature.

Depending on the moment when the ping flooding force the interruption, similar results have been obtained by modifying operands on the stack and local variables. What emerges from these different options is that a certain inaccuracy in the ping flooding phase is tolerable. Indeed, depending on the "breakpoint" location, an attacker with a good knowledge of the targeted application will often (not to say always) find a path to meet her objective.

This proof of concept demonstrates that such a threat should be taken in consideration when addressing the security of an embedded platform. Hereafter, we discuss this particular topic and tackle the issue of countermeasures designing.

5 Discussion on Protections

Making a product secure against any known attacks is not straightforward, as it requires a significant expertise in the field. This is particularly true in embedded technology when the security must coexist with restrictions of cost, performances and resources. Therefore the best way to suit all these requirements remains an optimisation of the security at the right protection level. This be can achieved thanks to a thorough vulnerability analysis. Regarding the attack introduced in this article, the developer should wonder if this attack is worth of being considered as a reasonable threat.

This attack is undoubtedly not easy to set up. However its apparent complexity should not conceal the potential consequences for an application. This statement is particularly true for the following reasons:

- The fault model turns out to be extremely powerful. Therefore most of the sensitive functions of an application may be defeated, even if they have been secured with care.
- A weak system may lead to an alteration of any hosted applications.
- The adequate protection is unlikely to be found in the application. As a result, an application with a thorough concern of security may be broken. It is then of the utmost importance that a system shows the evidence it is reliable and trustworthy.

Many ways can be explored to find efficient protections against this threat. Firstly it is worth of strengthening the scheduler to make sure it cannot be abused. The protection must be adapted to the rule enforced by the multithreaded system. Based on a time slice the scheduler of our Java Card makes use of a timer. By managing this timer appropriately in the handler in charge of the network requests, we have experimented that the Java Card withstands the attack.

The identification of the targeted instruction on the power consumption trace has also been an elementary step of our attack. Therefore, the difficulty to set up the attack increases with the difficulty to locate the instruction to attack. Techniques to harden the power consumption analysis such as described in [16, 17] would then stand as an additional barrier to circumvent for the attacker.

Another way consists of a strong isolation between the memory areas of different contexts. This includes the runtime environment area where the thread contexts are stored. Such an isolation may prevent the attacker to have access to the sensitive context. It is more or less difficult to achieve according to the systems. On a smartcard it may be interesting to take advantage of specific hardware features, such as a memory protection unit (MPU). This kind a protections enforces a strong isolation by the mean of hardware controls, which remain very difficult to overcome.

Lastly it may be worth of implementing some integrity controls on the contexts during the thread switch. As the control value must be prevented from being modified by an adversary, this may be achieved through a MAC verification using an internal symmetric key for instance. Before restoring a context, the scheduler would be in charge of checking that nothing has been tampered with

and could send an alarm if an inconsistency is found. This implementation has shown a great efficiency on the Java Card platform we used.

To meet with today's best practices it is assumed that the security should not rely on one single countermeasure. Therefore it is strongly recommended to combine at least two of these protections. As a conclusion, everyone has to bear in mind that finding the right compromise between security and performance is not obvious. This can be achieve by combining a high expertise in existing attacks with a strong experience in secure implementations.

6 Conclusion

This article introduces a novel attack exploiting a potential weakness of a multithreaded platform. An established breach would severely damage the security of any application running on this system. The principle lies on attempting to fool the scheduler. By this mean the attacker gets the ability to interrupt a sensitive code execution. By analogy he sets a breakpoint with the aim to subsequently modify the execution context and change the application behaviour.

The feasibility of this attack has been demonstrated on a smartcard implementing the JC3.0 specifications. Indeed this technology turned out to be a perfect target. With regards to the inherent constraints of embedded systems, it implements a relatively straightforward multithreading feature and offers network capabilities in a context of high security. As a result, we have shown how a strong authentication based on a signature may be bypassed.

Several ways of protecting an implementation have been introduced. All these techniques have shown a good level of efficiency on the Java Card. Now it is the developer's responsibility to figure out if this threat is worth of being considered.

With the growing complexity of some devices, several technologies are increasingly integrated together. This attack interestingly reveals that none of them must be neglected during the vulnerability analysis. Therefore any feature or functionality should be deemed as a potential door for an attack, even though they are not obviously related to the product security. The illustration on the Java Card with an exploitation of the multithreading and the network capability is meaningful.

Acknowledgement. The authors would like to thank Nicolas Morin, for his *practical* contribution to this work, and Christophe Giraud, for his most valuable comments on the different versions of this article.

References

1. Sun Microsystems Inc.: Runtime Environment Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
2. Sun Microsystems Inc.: Virtual Machine Specification, Java Card Platfokrm Version 3.0.1 Connected Edition (2009)

3. Sun Microsystems Inc.: Java Servlet Specification, Java Card Platform Version 3.0.1 Connected Edition (2009)
4. Anderson, R., Kuhn, M.: Tamper Resistance – a Cautionary Note. In: 2nd USENIX Workwhop on Electronic Commerce
5. Boneh, D., DeMillo, R., Lipton, R.: On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
6. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In: Smart Card Research and Advanced Application Conference (CARDIS 2004). LNCS, pp. 159–176. Springer, Heidelberg (2004)
7. Govindavajhala, S., Appel, A.W.: Using Memory Errors to Attack a Virtual Machine. In: SP 2003: Proceedings of the 2003 IEEE Symposium on Security and Privacy, Washington, DC, p. 154 (2003)
8. GlobalPlatform Inc.: GlobalPlatform Card Specification 2.2 (2006)
9. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley (1999)
10. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
11. Hogenboom, J., Mostowski, W.: Full memory read attack on a java card. In: 4th Benelux Workshop on Information and System Security Proceedings, WISSEC 2009 (2009)
12. Handley, M., Rescorla, E.: RFC 4732: Internet Denial-of-Service Considerations (2006)
13. CERT Coordination Center: Denial of Service Attacks. Technical report (1997), `http://www.cert.org/tech_tips/denial_of_service.html`
14. Vermoen, D., Witteman, M., Gaydadjiev, G.N.: Reverse Engineering Java Card Applets Using Power Analysis. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 138–149. Springer, Heidelberg (2007)
15. USB Implementers Forum, Inc.: Universal Serial Bus Communication Class Subclass Specification for Ethernet Emulation Model Devices (2005)
16. Shamir, A.: Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000. LNCS, vol. 1965, pp. 71–77. Springer, Heidelberg (2000)
17. Muresan, R., Gregori, S.: Protection Circuit against Differential Power Analysis Attacks for Smart Cards. IEEE Transactions on Computers 57, 1540–1549 (2008)
18. Sun Microsystems Inc.: Application Programming Interface, Java Card Platforms Version 3.0.1 Connected Edition (2009)

A Implementation

A.1 Array Forgery

The classes loaded wit the attacker's application :

```
public class ArrayContainer {
    byte[] array;
}
```

```
public class ForgeryContainer {
    Forgery f;
}

public class Forgery {
    int field_1, field_2;
}
```

The code within the attacker's application to forge the array in the volatile memory :

```
ArrayContainer ac = new ArrayContainer();
ac.array = new byte[1];
ForgeryContainer fc = (ForgeryContainer) (Object) ac;
fc.f.field_1 = 0x100; // set the length of ac.array to 256
fc.f.field_2++;       // increment the memory pointer of ac.array
// Access to memory through ac.array[i]
```

A.2 Request Flooding Validation Thread

The run method of the thread used to validate the influence of communication :

```
public void run() {
    i = 0;
    startTime = System.currentTimeMillis();
    while ((System.currentTimeMillis() - startTime) < TIME_BOUND){
        i++;
    }
}
```

The Python ping flooder *:*

```
def flood(host, url, delay, socket, ID, count, ping_delay)
    # Send request to target application
    conn = httplib.HTTPConnection(host)
    conn.request("GET", url)

    # Wait
    time.sleep(delay)

    # Send ping flood
    for i in xrange(count):
        send_ping(ID, socket, host)
        time.sleep(ping_delay)
```