CHAPTER

18

API Attacks

One is happenstance; twice is coincidence; but three times is enemy action.

— Goldfinger

Simplicity is the ultimate sophistication.

— Leonardo Da Vinci

18.1 Introduction

Many supposedly secure devices have some kind of application programming interface, or API, that untrustworthy people and processes can call in order to get some task performed.

- A bank's server will ask an attached hardware security module 'Here's a customer account number and PIN, with the PIN encrypted using the key we share with VISA. Is the PIN correct?'
- If you enable javascript, then your browser exposes an application programming interface — javascript — which the owners of websites you visit can use to do various things.
- A secure operating system may limit the calls that an application program can make, using a reference monitor or other wrapper to enforce a policy such as preventing information flow from High to Low.

The natural question to ask is whether it's safe to separate tasks into a trusted component and a less trusted one, and it's recently been realised that the answer is very often no. Designing security APIs is a very hard problem indeed.

API security is related to some of the problems I've already discussed. For example, multilevel secure systems impose flow controls on static data, while a security API is often trying to prevent some information flow in the presence of tight and dynamic interaction — a much harder problem. It's also related to protocol security: as I will discuss shortly, bank security modules are often insecure because of interactions between a number of different protocols that they support. It touches on software security: the javascript implementation in Firefox lets calling programs scan all variables set by existing plugins — which may compromise your privacy by leaking information about your browsing habits and web mail usage [543]. The javascript developers just didn't think to have a separate sandbox for each visited website. Indeed, the most common API failure mode is that transactions that are secure in isolation become insecure in combination, whether because of application syntax, feature interaction, slow information leakage or concurrency problems.

There are many other examples. In the context of embedded systems, for example, I discussed prepayment electricity meters; the token vending machines use tamper-resistant security modules to protect keys and value counters. There, an attack was to change the electricity tariff to the lowest possible value and issue tokens that entitled recipients to power for much less than its market value. The design error was to not bind in tariff reporting into the end-to-end protocol design.

A potentially critical example is 'Trusted Computing'. This initiative has put a TPM chip for secure crypto key storage on most of the PC and Mac motherboards shipping today. The plan, according to Microsoft, is that future applications will have a traditional 'insecure' part running on top of Windows as before, and also a 'secure' part or NCA that will run on top of a new security kernel known as the Nexus, which will be formally verified and thus much more resistant to software attacks. The Nexus and the NCA will guard crypto keys and other critical variables for applications. The question this then raises is now the interface between the application and the NCA is to be protected.

In short, whenever a trusted computer talks to a less trusted one, the language they use is critical. You have to expect that the less trusted device will try out all sorts of unexpected combinations of commands in order to trick the more trusted one. How can we analyse this systematically?

18.2 API Attacks on Security Modules

We have learned a lot about API security from failures of the hardware security modules used by banks to protect PINs and crypto keys for ATM networks. In 1988, Longley and Rigby identified the importance of separating key types while doing work for security module vendor Eracom [811]. In 1993,

we reported a security flaw that arose from a custom transaction added to a security module [69]. However the subject really got going in 2000 when I started to think systematically about whether there might be a series of transactions that one could call from a security module that would break its security [52]. I asked: 'So how can you be sure that there isn't some chain of 17 transactions which will leak a clear key?' Looking through the manuals, I discovered the following vulnerability.

18.2.1 The XOR-To-Null-Key Attack

Hardware security modules are driven by transactions sent to them by the host computers to which they are attached. Each transaction typically consists of a command, a serial number, and several data items. The response contains the serial number and several other data items. The security module contains a number of master keys that are kept in tamper-responding memory, which is zeroized if the device is opened. However, there is often not enough storage in the device for all the keys it might have to use, so instead keys are stored encrypted outside the device. Furthermore, the way in which these working keys are encrypted provides them with a type system. For example, in the security modules provided by VISA, a key used to derive a PIN from an account number (as described in section 10.4.1) is stored encrypted under a particular pair of master DES keys.

Among the VISA security module's transactions can be found support for generating a *Terminal Master Key* for an ATM. You'll recall from Chapter 10 that ATM security is based on dual control, so a way had to be found to generate two separate keys that could be carried from the bank to the ATM, say by the branch manager and the branch accountant, and entered into it at a keypad. The VISA device thus had a transaction to generate a key component and print out its clear value on an attached security printer. It also returned its value to the calling program, encrypted under the relevant master key KM, which was kept in the tamper-resistant hardware:

$$\text{VSM} \rightarrow \text{printer: } KMT_i$$
$$\text{VSM} \rightarrow \text{host: } \{KMT_i\}_{KM}$$

It also had another transaction that will combine two such components to produce a terminal key:

$$\text{Host} \rightarrow \text{VSM: } \{KMT_1\}_{KM}, \{KMT_2\}_{KM}$$
$$\text{VSM} \rightarrow \text{host: } \{KMT_1 \oplus KMT_2\}_{KM}$$

The idea was that to generate a terminal key for the first time, you'd use the first of these transactions twice followed by the second. Then you'd have $KMT = KMT_1$ exclusive-or KMT_2. However, there was nothing to stop the

programmer taking any old encrypted key and supplying it twice in the second transaction, resulting in a known terminal key (the key of all zeroes, as the key is exclusive-or'ed with itself):

$$\text{Host} \longrightarrow \text{VSM: } \{KMT_1\}_{KM}, \{KMT_1\}_{KM}$$
$$\text{VSM} \longrightarrow \text{host: } \{KMT_1 \oplus KMT_1\}_{KM}$$

So now we have managed to insert a known key into the system. What can be done with it? Well, the module also had a transaction to encrypt any key supplied encrypted under *KM*, under any other key that itself is encrypted under *KM*. Sounds complicated? Well, that's where things break. The purpose of this odd transaction was to enable a bank to encrypt its PIN verification key under a terminal master key, so it could be sent out to an ATM for offline PIN verification. What's more, the key type 'terminal master key' and the key type 'PIN verification key' were the same. The effect of this was drastic. A programmer can retrieve the bank's PIN verification key — which is also kept outside the security module, encrypted with *KM*, and have it encrypted under the zero key that he now has inserted into the system:

$$\text{Host} \longrightarrow \text{VSM: } \{0\}_{KM}, \{PIN\}_{KM}$$
$$\text{VSM} \longrightarrow \text{host: } \{PIN\}_0$$

The programmer can now decrypt the PIN verification key — the bank's crown jewels — and can work out the PIN for any customer account. The purpose of the security module has been completely defeated; the bank might as well have just worked out PINs using encryption software, or kept them in clear on its database.

The above attack went undiscovered for a long time because it's a bit hard to understand the implications of *'a transaction to encrypt any key supplied encrypted under KM, under any other key that itself is encrypted under KM'*. It was just not clear what the various types of key in the device were suppose to do, and what security properties the type system had to have in order to achieve its goals. In fact, there seemed to be no formal statement of the protection goals at all; the module had simply evolved from earlier, simpler designs as banks asked for more features.

The next attack was found using formal methods. My student Mike Bond built a formal model of the key types used in the device and immediately discovered another flaw. The key type 'communications key' is used for MAC keys, which have the property that you can input a MAC key in the clear and get it encrypted by — you guessed it — 'any key that itself is encrypted under *KM*'. So here was another way in which you could get a known key into the system. You could put in an account number into the system, pretending it's a MAC key, and get it encrypted with the PIN verification key — this gives you

the customer PIN directly. (Confused? Initially everyone was — modern APIs are just too complicated for bugs to be evident on casual inspection. Anyway, the full details are at [65].)

18.2.2 The Attack on the 4758

We next found a number of cryptographic attacks on the API of the VISA module and on its IBM equivalent, the 4758 [641]. For example, they both generate 'check values' for keys — a key identifier calculated by encrypting a string of zeroes under the key. This opens up a birthday attack, also known as a time-memory tradeoff attack. Suppose you want to crack a DES key of a type that you can't extract from the device. Now DES keys can be broken with about 2^{55} effort, but that's a lot of work. However, in the security module architecture it's generally enough to crack any key of a given type, as data can be converted back and forth between them. You're also able to generate large numbers of keys of any given type — several hundred a second.

The attack therefore goes as follows.

1. Generate a large number of terminal master keys, and collect the check value of each.
2. Store all the check values in a hash table.
3. Perform a brute force search, by guessing a key and encrypting the fixed test pattern with it.
4. Compare the resulting check value against all the stored check values by looking it up in the hash table (an $O(1)$ operation).

With a 2^{56} keyspace, an attacker who can generate 2^{16} target keys (which can be done over lunchtime), a target key should be hit by luck with roughly $2^{56}/2^{16} = 2^{40}$ effort (which you can do in a week or so on your PC). This is also called a 'meet-in-the-middle' attack, reflecting the meeting of effort spent by the HSM generating keys and effort spent by the brute-force search checking keys.

Within a short space of time, Mike Bond had come up with an actual attack on the 4758. This really shook the industry, as the device had been certified to FIPS 140-1 level 4; in effect the US government had said it was unbreakable. In addition to the meet-in-the-middle attack, he used a further obscure design error, key replication.

As DES became vulnerable to keysearch during the 1980s, financial institutions started migrating systems to two-key triple-DES: the block was encrypted with the left key, decrypted with the right key and then encrypted with the left key once more. This piece of cleverness gave a backward compatibility

mode: set the left key equal to the right key, and the encryption reverts to single-DES. The 4758 compounded the cleverness by storing left keys and right keys separately. IBM took precautions to stop them being confused — left keys and right keys were encrypted differently, giving them different types — but failed to bind together the two halves.

So an attacker could use the meet-in-the-middle trick to get two single DES keys with known values inside a 4758, then swap the left and right halves to get a known true triple-DES key. This could then be used to export other valuable keys. Several variants of this attack are described in [202]; the attack was actually implemented and demonstrated on prime-time television [306].

18.2.3 Multiparty Computation, and Differential Protocol Attacks

The next set of attacks on security module APIs was initiated by Jolyon Clulow in 2003 [309], and they depend on manipulating the details of the application logic so as to leak information.

His first attack exploited error messages. One of the PIN block formats in common use combines the PIN and the account number by exclusive-or and then encrypts them; this is to prevent attacks such as the one I described in section 10.4.2 where the encrypted PIN isn't linked to the customer account number. However (as with combining keys) it turned out that exclusive-or was a bad way to do this. The reason was this: if the wrong account number was sent along with the PIN block, the device would decrypt the PIN block, xor in the account number, discover (with reasonable probability) that the result was not a decimal number, and return an error message. The upshot was that by sending a series of transactions to the security module that had the wrong account number, you could quickly work out the PIN.

An even simpler attack was then found by Mike Bond and Piotr Zielinski. Recall the method used by IBM (and most of the industry) to generate PINs, as in Figure 18.1:

Account number *PAN*:	`8807012345691715`
PIN key *KP*:	`FEFEFEFEFEFEFEFE`
Result of DES $\{PAN\}_{KP}$:	`A2CE126C69AEC82D`
$\{N\}_{KP}$ decimalized:	`0224126269042823`
Natural PIN:	`0224`
Offset:	`6565`
Customer PIN:	`6789`

Figure 18.1: IBM method for generating bank card PINs

The customer account number is encrypted using the PIN verification key, yielding a string of 16 hex digits. The first four are converted to decimal digits for use as the PIN using a user-supplied function: the *decimalisation table*. This gives the natural PIN; an offset is then added whose function is to enable the customer to select a memorable PIN. The customer PIN is the natural PIN plus the offset. The most common decimalisation table is `012345689012345`, which just takes the DES output modulo 10.

The problem is that the decimalisation table can be manipulated. If we set the table to all zeros (i.e., `0000000000000000`) then a PIN of '0000' will be generated and returned in encrypted form. We then repeat the call using the table `1000000000000000`. If the encrypted result changes, we know that the DES output contained a 0 in its first four digits. Given a few dozen suitably-chosen queries, the value of the PIN can be found. Since the method compares repeated, but slightly modified, runs of the same protocol, we called this attack *differential protocol analysis*.

The industry's initial response was to impose rules on allowable decimalisation tables. One vendor decreed, for example, that a table would have to have at least eight different values, with no value occurring more than four times. This doesn't actually cut it (try `0123456789012345`, then `1123456789012345`, and so on). The only real solution is to rip out the decimalisation table altogether; you can pay your security-module vendor extra money to sell you a machine that's got you own bank's decimalisation table hard-coded.

At a philosophical level, this neatly illustrates the difficulty of designing a device that will perform a secure multiparty computation — where a computation has to be performed using secret information from one party, and some inputs that can be manipulated by a hostile party [64][1]. Even in this extremely simple case, it's so hard that you end up having to abandon the IBM method of PIN generation, or at least nail down its parameters so hard that you might as well not have made those parameters tweakable in the first place.

At a practical level, it illustrates one of the main reasons APIs fail. They get made more and more complex, to accommodate the needs of more and more diverse customers, until suddenly there's an attack.

18.2.4 The EMV Attack

You'd have thought that after the initial batch of API attacks were published in 2001, security module designers would have started being careful about adding new transactions. Not so! Again and again, the banking industry has demanded the addition of new transactions that add to the insecurity.

[1]We came across this problem in Chapter 9 where we discussed active attacks on inference control mechanisms.

An interesting recent example is a transaction ordered by the EMV consortium to support secure messaging between a smartcard and a bank security module. The goal is that when a bank card appears in an online transaction, the bank can order it to change some parameter, such as a new key. So far so good. However, the specification had a serious error, which has appeared in a number of implementations [12].

The transaction `Secure Messaging For Keys` allows the server to command the security module to encrypt a text message, followed by a key, with a key of a type for sharing with bank smartcards. The encryption can be in CBC or ECB mode, and the text message can be of variable length. This lets an attacker choose a message length so that just one byte of the target key crosses the boundary of an encryption block. That byte can then be determined by sending a series of messages that are one byte longer, where the extra byte cycles through all possible values until the key byte is found. The attacker can then attack each of the other key bytes, one after another. Any exportable key can be extracted from the module in this way. (There is also a transaction `Secure Messaging For PINs`, but who needs to extract PINs one by one if you can rip off all the master keys?)

To sum up: the security modules sold to the banking industry over the last quarter century were almost completely insecure, because the APIs were very badly designed. At one time or another, we found an attack on at least one version of every security module on the market. In time the vendors stopped or mitigated most of these attacks by shipping software upgrades. However, the customers — the banking industry — keep on thinking up cool new things to do with payment networks, and these keep on breaking the APIs all over again. The typical attacks involve multiple transactions, with the technical cause being feature interaction (with particularly carelessly designed application features) or slow information leakage. The root cause, as with many areas of our field, is featuritis. People make APIs so complex that they break, and the breaks aren't at all obvious.

18.3 API Attacks on Operating Systems

A second class of API attacks involve concurrency, and are well illustrated by vulnerabilities found by Robert Watson in system call wrappers [1325].

System call wrappers are used to beef up operating systems security; the reference monitors discussed in Chapter 8 are an example, and anti-virus software is another. Wrappers intercept calls made by applications to the operating system, parse them, audit them, and may pass them on with some modification: for example, a Low process that attempts to access High data may get diverted to dummy data or given an error message. There are various frameworks available, including Systrace, the Generic Software

Wrapper Toolkit (GSWTK) and CerbNG, that enable you to write wrappers for common operating systems. They typically execute in the kernel's address space, inspect the enter and exit state on all system calls, and encapsulate only security logic.

When John Anderson proposed the concept of the reference monitor in 1972, he stipulated that it should be tamper-proof, nonbypassable, and small enough to verify. On the face of it, today's wrappers are so — until you start to think of time. Wrappers generally assume that system calls are atomic, but they're not; modern operating system kernels are very highly concurrent. System calls are not atomic with respect to each other, now are they so with respect to wrappers. There are many possibilities for two system calls to race each other for access to shared memory, and this gives rise to *time-of-check-to-time-of-use* (TOCTTOU) attacks of the kind briefly mentioned in Chapter 6.

A typical attack is to race on user-process memory by getting the kernel to sleep at an inconvenient time, for example by a page fault. In an attack on GSWTK, Watson calls a path whose name spills over a page boundary by one byte, causing the kernel to sleep while the page is fetched; he then replaces the path in memory. It turns out that the race windows are large and reliable.

Processors are steadily getting more concurrent, with more processors shipped in each CPU chip as time passes, and operating systems are optimised to take advantage of them. This sort of attack may become more and more of a problem; indeed, as code analysis tools make stack overflows rarer, it may well be that race conditions will become the attack of choice.

What can be done to limit them? The only real solution is to rewrite the API. In an ideal world, operating systems would move to a message-passing model, which would eliminate (or at least greatly reduce) concurrency issues. That's hardly practical for operating system vendors whose business models depend on backwards compatibility. A pragmatic compromise is to build features into the operating system specifically to deal with concurrency attacks; Linux Security Modules do this, as does Mac OS/X 10.5 (based on TrustedBSD, on which Watson worked).

In short, the APIs exposed by standard operating systems have a number of known weaknesses, but the wrapper solution, of interposing another API in front of the vulnerable API, looks extremely fragile in a highly concurrent environment. The wrapper would have to understand the memory layout fairly completely to be fully effective, and this would make it as complex (and vulnerable) as the operating system it was trying to protect.

18.4 Summary

Interfaces get richer, and dirtier, and nastier, over time. Interface design flaws are widespread, from the world of cryptoprocessors through sundry embedded systems right through to antivirus software and the operating

system itself. Wherever trusted software talks to less trusted software, the interface is likely to leak more than the designer of the trusted software intended.

Failures tend to arise from complexity. I've discussed two main case histories — cryptoprocessors, which accumulated transactions beyond the designers' ability to understand their interactions, and system call wrappers, which try to audit and filter the calls made to an operating system's API. At the level of pure computer science, we could see the former as instances of the composition problem or the secure multiparty computation problem, and the latter as concurrency failures. However, these are industrial-scale systems rather than blackboard systems. A security module may have hundreds of transactions, with a fresh batch being added at each upgrade. At the application level, many of the failures can be seen as feature interactions. There are also specific failures such as the slow leakage of information from poorly designed cryptosystems, which would not be serious in the case of a single transaction but which become fatal when an opponent can commandeer a server and inject hundreds of transactions a second.

API security is already important, and becoming more so as computers become more complex, more diverse and more concurrent. If Microsoft ships the original mechanisms promised for Trusted Computing with a future release of Vista, then APIs will become more critical still — application developers will be encouraged to write code that contains a 'more trusted' NCA and a 'less trusted' normal application. Even professionals who work for security module companies ship APIs with serious bugs; what chance is there that NCAs will provide value, if everyone starts designing their security APIs?

What can be done? Well, one of the lessons learned is that a 'secure' processor isn't, unless the API is simple enough to understand and verify. If you're responsible for a bank's cryptography, the prudent thing to do is to have a security module adapted for your needs so that it contains only the transactions you actually need. If this is too inconvenient or expensive, filter the transactions and throw away all but the essential ones. (Do read the research literature carefully before deciding which transactions are essential!)

As Watson's work shows, complex APIs for highly-concurrent systems probably cannot be fixed in this way. If you've got critical applications that depend on such a platform, then maybe you should be making migration plans.

Finally, there will probably be a whole host of API issues with interacting applications. As the world moves to web services that start to talk to each other, their APIs are opened up to third-party developers — as has just happened, for example, with Facebook. Complexity alone is bad enough; I'll discuss in Chapter 23 how social-networking sites in particular are pushing complexity limits in security policy as they try to capture a significant subset of human

social behaviour. And managing the evolution of an API is one of the toughest jobs in security engineering (I'll come to this in Chapter 25). Combine this with the fact that economic pressures push web applications toward unsafe defaults, such as making everything searchable, and the strong likelihood that not all third-party developers will be benevolent — and it should be clear that we can expect problems.

Research Problems

The computer science approach to the API security problem has been to try to adapt formal-methods tools to prove that interfaces are safe. There is a growing literature on this, but the methods can still only tackle fairly simple APIs. Verifying an individual protocol is difficult enough, and the research community spent much the 1990s learning how to do it. Yet a protocol might consist of perhaps 2–5 messages, while a cryptoprocessor might have hundreds of transactions.

An alternative approach, as in the world on protocols, is to try to come up with robustness principles to guide the designer. As in that field, robustness is to some extent about explicitness. Checking that there isn't some obscure sequence of transactions that breaks your security policy is hard enough; when your policy isn't even precisely stated it looks impossible.

Robustness isn't everything, though. At the tactical level, the API security story has taught us a number of things. Atomicity really does matter (together with consistency, isolation and durability, the other desirable attributes of transaction-processing systems). At an even lower level, we've seen a couple of good reasons why it's a really dumb idea to use exclusive-or to combine keys or PINs; no-one understood this before. What other common design practices should we unlearn?

Further Reading

To learn more about API security, you should first read our survey papers [63, 64, 65] as well as Robert Watson's paper on concurrency attacks [1325]. There is now an annual conference, the International Workshop on Analysis of Security APIs, where you can catch up with the latest research.