CHAPTER

1

Why Care About Database Security?

In the introduction, we discussed the reasons why we consider database security to be important. In this chapter, we provide a brief overview of several broad categories of security issues, with a few specific details and some discussion of general defenses. We also briefly discuss how to go about finding security flaws in database systems. Before we do so, we should discuss some emerging trends in database security.

In recent years, with the explosion in web-based commerce and information systems, databases have been drawing ever closer to the network perimeter. This is a necessary consequence of doing business on the Web — you need your customers to have access to your information via your web servers, so your web servers need to have access to your databases. Databases that were previously accessible only via several insulating layers of complex business logic are now directly accessible from the much more fluid — and much less secure — web application environment. The result of this is that the databases are closer to the attackers. With the constant march toward a paperless business environment, database systems are increasingly being used to hold more and more sensitive information, so they present an increasingly valuable target. In recent years, database vendors have been competing with each other to provide the most feature-rich environment they can, with most major systems supporting XML, web services, distributed replication, operating system integration, and a host of other useful features. To cap all of this, the legislative burden in terms of corporate security is increasing, with HIPAA, SOX, GLBA, and California Senate Bill No. 1386 imposing an ever-increasing pressure on companies to ensure that their networks are compliant.

So why care about database security? Because your databases are closer to the attacker, present a more valuable target, have more features to configure, and are more closely regulated than they have ever been before.

Which Database Is the Most Secure?

All of the databases we cover in this volume have had serious security flaws at some point. Oracle has published 69 security alerts on its "critical patch updates and security alerts" page — though some of these alerts relate to a large number of vulnerabilities, with patch 68 alone accounting for somewhere between 50 and 100 individual bugs. Depending on which repository you search, Microsoft SQL Server and its associated components have been subject to something like 36 serious security issues — though again, some of these patches relate to multiple bugs. According to the ICAT metabase, DB2 has had around 20 published security issues — although the authors of this book have recently worked with IBM to fix a further 13 issues. MySQL has had around 25 issues; Sybase ASE is something of a dark horse with a mere 2 published vulnerabilities. PostgreSQL has had about a dozen. Informix has had about half a dozen, depending on whose count you use.

The problem is that comparing these figures is almost entirely pointless. Different databases receive different levels of scrutiny from security researchers. To date, Microsoft SQL Server and Oracle have probably received the most, which accounts for the large number of issues documented for each of those databases. Some databases have been around for many years, and others are relatively recent. Different databases have different kinds of flaws; some databases are not vulnerable to whole classes of problems that might plague others. Even defining "database" is problematic. Oracle bundles an entire application environment with its database server, with many samples and prebuilt applications. Should these applications be considered a part of the database? Is Microsoft's MSDE a different database than SQL Server? They are certainly used in different ways and have a number of differing components, but they were both subject to the UDP Resolution Service bug that was the basis for the "Slammer" worm.

Even if we were able to determine some weighted metric that accounted for age, stability, scrutiny, scope, and severity of published vulnerabilities, we would still be considering only "patchable" issues, rather than the inherent security features provided by the database. Is it fair to directly compare the comprehensive audit capabilities of Oracle with the rather more limited capabilities of MySQL, for instance? Should a database that supports securable views be considered "more secure" than a database that doesn't implement that abstraction? By default, PostgreSQL is possibly the most security-aware database available — but you can't connect to it over the network unless you explicitly enable that functionality. Should we take default configurations into account? The list of criteria is almost endless, and drawing any firm conclusions from it is extremely dangerous.

Ultimately, the more you know about a system, the better you will be able to secure it — up to a limit imposed by the features of that system. It isn't true to say, however, that the system with the most features is the most secure because the more functionality a system has, the more target surface there is for an attacker to abuse. The point of this book is to demonstrate the strengths and weaknesses of the various database systems we're discussing, not — most emphatically not — to determine which is the "most secure."

In the end, the most secure database is the one that you know the most about.

The State of Database Security Research

Before we can discuss the state of database security research, we should first define what we mean by the term. In general, when we use the phrase "database security research" we tend to mean research into specific, practical flaws in the security of database systems. We do not mean research into individual security incidents or discussions of marketing-led accreditation or certification efforts. We don't even mean academic research into the underlying abstractions of database security, such as field-, row-, and object-level security, or encryption, or formal protocol security analysis — though the research we are talking about may certainly touch on those subjects. We mean research relating to discoveries of real flaws in real systems.

So with that definition in mind, we will take a brief tour of recent — and not so recent — discoveries, and attempt to classify them appropriately.

Classes of Database Security Flaws

If you read about specific security flaws for any length of time, you begin to see patterns emerge, with very similar bugs being found in entirely different products. In this section, we attempt to classify the majority of known database security issues into the following categories:

- Unauthenticated Flaws in Network Protocols
- Authenticated Flaws in Network Protocols
- Flaws in Authentication Protocols
- Unauthenticated Access to Functionality

- Arbitrary Code Execution in Intrinsic SQL Elements
- Arbitrary Code Execution in Securable SQL Elements
- Privilege Elevation via SQL Injection
- Local Privilege Elevation Issues

So we begin with arguably the most dangerous class of all — unauthenticated flaws in network protocols. By this we mean buffer overflows, format string bugs, and so on, in the underlying network protocols used by database systems.

Unauthenticated Flaws in Network Protocols

Arguably the most famous bug in this class is the bug exploited by the SQL Server "Slammer" worm. The SQL Server Resolution Service operates over a UDP protocol, by default on port 1434. It exposes a number of functions, two of which were vulnerable to buffer overflow issues (CAN-2002-0649). These bugs were discovered by David Litchfield of NGS. Another SQL Server problem in the same category was the "hello" bug (CAN-2002-1123) discovered by Dave Aitel of Immunity, Inc., which exploited a flaw in the initial session setup code on TCP port 1433.

Oracle has not been immune to this category — most recently, David Litchfield found an issue with environment variable expansion in Oracle's "extproc" mechanism that can be exploited without a username or password (CAN-2004-1363). Chris Anley of NGS discovered an earlier flaw in Oracle's extproc mechanism (CAN-2003-0634) that allowed for a remote, unauthenticated buffer overflow. Mark Litchfield of NGS discovered a flaw in Oracle's authentication handling code whereby an overly long username would trigger an exploitable stack overflow (CAN-2003-0095).

David Litchfield also found a flaw in DB2's JDBC Applet Server (no CVE, but bugtraq id 11401) that allows a remote, unauthenticated user to trigger a buffer overflow.

In general, the best way to defend yourself against this class of problem is first, to patch. Second, you should attempt to ensure that only trusted hosts can connect to your database servers, possibly enforcing that trust through some other authentication mechanism such as SSH or IPSec. Depending on the role that your database server is fulfilling, this may be tricky.

Another possibility for defense is to implement an Intrusion Detection System (IDS) or an Intrusion Prevention System (IPS). These kinds of systems have been widely discussed in security literature, and are of debatable value. Although an IDS can (sometimes) tell you that you have been compromised, it won't normally prevent that compromise from happening. Signature-based

IDS systems are only as strong as their signature databases and in most cases signatures aren't written by people who are capable of writing exploits, so many loopholes in the signatures get missed.

"True anomaly" IDS systems are harder to bypass, but as long as you stick to a protocol that's already in use, and keep the exploit small, you can usually slip by. Although some IDS systems are better than others, in general you need an IDS like you need someone telling you you've got a hole in the head. IDS systems will certainly stop dumber attackers, or brighter attackers who were unlucky, so they may be worthwhile provided they complement — and don't replace — skilled staff, good lockdown, and good procedures.

IPS systems, on the other hand, do prevent some classes of exploit from working but again, every IPS system the authors have examined can be bypassed with a little work, so your security largely depends on the attacker not knowing which commercial IPS you're using. Someone may bring out an IPS that prevents all arbitrary code execution attacks at some point, which would be a truly wonderful thing. Don't hold your breath waiting for it, though.

Authenticated Flaws in Network Protocols

There are substantially fewer bugs in this category. This may reflect a reduced focus on remote, authenticated bugs versus remote, unauthenticated bugs among the security research community, or it may be sheer coincidence.

David Litchfield found a flaw in DB2 for Windows (CAN-2004-0795) whereby a remote user could connect to the DB2REMOTECMD named pipe (subject to Windows authentication) and would then be able to execute arbitrary commands with the privilege of the db2admin user, which is normally an "Administrator" account.

David discovered another flaw in DB2 in this category recently, relating to an attacker specifying an overly long locale LC_TYPE. The database applies this after the user authenticates, triggering the overflow.

There are several other bugs that debatably fall into this category, normally relating to web application server components; because we're focusing on the databases themselves we'll gloss over them.

In general the best way to protect yourself against this category of bugs is to carefully control the users that have access to your databases; a strong password policy will help — as long as you're not using plaintext authentication protocols (we discuss this more later). Auditing authenticated users is also a good idea for a number of reasons; it might give you a heads-up if someone is trying to guess or brute-force a password, and if you do have an incident, at least you have somewhere to start looking.

Flaws in Authentication Protocols

Several database systems have plaintext authentication protocols, by which we mean authentication protocols in which the password is passed "on the wire" in a plaintext or easily decrypted format. In a default configuration (that Sybase warns against, but which we have still seen in use) Sybase passes passwords in plaintext. By default, Microsoft SQL Server obfuscates passwords by swapping the nibbles (4-bit halves of a byte) and XORing with 0xA5. In both of these cases, the vendors warn against using the plaintext versions of their authentication protocols and provide strong, encrypted mechanisms that are relatively easy to deploy — but the defaults are still there, and still dangerous.

MySQL has historically had a number of serious problems with its authentication protocol. Although the protocol isn't plaintext, the mathematical basis of the authentication algorithm prior to version 4.1 was called into question by Ariel Waissbein, Emiliano Kargieman, Carlos Sarraute, Gerardo Richarte, and Agustin Azubel of CORE SDI (CVE-2000-0981). Their paper describes an attack in which an attacker that can observe multiple authentications is quickly able to determine the password hash.

A further conceptual problem with the authentication protocol in MySQL prior to version 4.1 is that the protocol only tests knowledge of the password hash, not the password itself. This leads to serious problems if a user is able to somehow determine another user's password hash — and MySQL has been subject to a number of issues in which that was possible.

Robert van der Meulen found an issue (CVE-2000-0148) in MySQL versions prior to 3.23.11 whereby an attacker could authenticate using only a single byte of the expected response to the server's challenge, leading to a situation whereby if you knew a user's username, you could authenticate as that user in around 32 attempts.

Chris Anley recently found a very similar problem in MySQL (CAN-2004-0627) whereby a user could authenticate using an empty response to the server's challenge, provided he or she passed certain flags to the remote server.

This category of bugs is almost as dangerous as the "unauthenticated flaws in network protocols" category, because in many cases the traffic simply looks like a normal authentication. Attackers don't need to exploit an overflow or do anything clever, they simply authenticate without necessarily needing the password — or if they've been able to sniff the password, they just authenticate.

The best defense against this kind of bug is to ensure that your database patches are up-to-date, and that you don't have any plaintext authentication mechanisms exposed on your databases. If your DBMS cannot support encrypted authentication in your environment, you could use IPSec or SSH to provide an encrypted tunnel. MySQL provides explicit guidelines on how to do this in its documentation, though recent versions of MySQL allow authentication to take place over an SSL-encrypted channel.

Unauthenticated Access to Functionality

Some components associated with databases permit unauthenticated access to functionality that should really be authenticated. As an example of this, David Litchfield found a problem with the Oracle 8 and 9i TNS Listener, whereby a remote, unauthenticated user could load and execute an arbitrary function via the "extproc" mechanism (CVE-2002-0567). The function can have any prototype, so the obvious mode of attack is to load the libc or msvcrt library (depending upon the target platform) and execute the "system" function that allows an attacker to execute an arbitrary command line. The command will be executed with the privileges of the user that the database is running as — "oracle" on UNIX systems, or the local system user on Windows.

Recently, David Litchfield disclosed an issue that allows any local user to execute commands in the security context of the user that Oracle is running as (CAN-2004-1365). This bug works in exactly the same way as the bug listed earlier (CVE-2002-0567), except that it takes advantage of the implicit trust that extproc places in the local host. Oracle does not consider this to be a security issue (!) but we would caution you not to allow users to have shells on Oracle servers without seriously considering the security ramifications. Clearly, allowing a user to have a shell on a database server is dangerous anyway, but in this particular case there is a known, documented vector for attack that the vendor will not fix.

There is a whole class of attacks that can be carried out on unsecured Oracle TNS Listeners, including writing arbitrary data to files, that we cover later in the Oracle chapters of this book — Oracle recommends that a Listener password be set, but it is not unusual to find servers where it hasn't been.

Arbitrary Code Execution in Intrinsic SQL Elements

This class of buffer overflow applies to buffer overflow and format string bugs in elements of the database's SQL grammar that are not subject to the usual access control mechanisms (GRANT and REVOKE). This class is rather more of a threat than it might initially appear, since these bugs can normally be triggered via SQL injection problems in Internet-facing web applications. A well-written exploit for a bug in this class could take a user from the Internet into administrative control of your database server in a single step.

A good example of this kind of thing in Microsoft SQL Server was the pwdencrypt overflow discovered by Martin Rakhmanoff (CAN-2002-0624). This was a classic stack overflow in a function used by SQL Server to encrypt passwords.

An example of a format string bug in this category was the RAISERROR format string bug discovered in SQL Server 7 and 200 by Chris Anley (CAN-2001-0542).

Oracle has been subject to several bugs in this category — although it is normally possible to revoke access to Oracle functions, it can be somewhat problematic. Mark Litchfield discovered that the TIME_ZONE session parameter, and NUMTOYMINTERVAL, NUMTODSINTERVAL, FROM_TZ functions are all subject to buffer overflows that allow an attacker to execute arbitrary code.

David Litchfield discovered that the DB2 "call" mechanism was vulnerable to a buffer overflow that can be triggered by any user (no CVE-ID, but bugtraq ID 11399).

Declaring a variable with an overly long data type name in Sybase ASE versions prior to 12.5.3 will trigger an overflow.

Most databases have flaws in this category, simply because parsing SQL is a hard problem. Developers are likely to make mistakes, and since parsing code can be so convoluted, it can be hard to tell whether or not code is secure.

The best defense against this category of bugs is to patch. Allowing untrusted users to influence SQL queries on the database server can also be a bad idea; most organizations are aware of the threat posed by SQL injection but it is still present in a sizeable proportion of the web applications that we audit. This category of bugs, perhaps more so than any other, is a great argument for ensuring that your patch testing and deployment procedures are as slick as they can be.

Arbitrary Code Execution in Securable SQL Elements

In a slightly less severe category than the intrinsic function overflows, we have the set of overflow and format string bugs that exist in functions that can be subject to access controls. The interesting thing about this category is that, although the risk from these problems can be mitigated by revoking permissions to the objects in question, they are normally accessible by default.

Several bugs in this category have affected Microsoft SQL Server — Chris Anley discovered buffer overflows in the extended stored procedures xp_setsqlsecurity (CAN-2000-1088), xp_proxiedmetadata (CAN-2000-1087), xp_printstatements (CAN-2000-1086), and xp_peekqueue (CAN-2000-1085). David Litchfield discovered buffer overflows in the xp_updatecolvbm (CAN-2000-1084), xp_showcolv (CAN-2000-1083), xp_enumresultset (CAN-2000-1082), and xp_displayparamstmt (CAN-2000-1081) extended stored procedures.

Mark Litchfield discovered a buffer overflow in the BULK INSERT statement in SQL Server (CAN-2002-0641); by default the owner of a database can execute this statement but a successful exploit will normally confer administrative privileges on the target host.

David Litchfield discovered an overflow in Oracle's CREATE DATABASE LINK statement (CAN-2003-0222); by default CREATE DATABASE LINK privilege is assigned to the CONNECT role — though low-privileged accounts such as SCOTT and ADAMS can normally create database links.

Patching is the best defense against this category of bugs, though a good solid lockdown will eliminate a fair portion of them. The difficulty with removing "default" privileges is that often there are implicit dependencies — system components might depend on the ability to execute the stored procedure in question, or some replication mechanism might fail if a given role has its permissions revoked. Debugging these issues can sometimes be tricky. It is definitely worth investing some time and effort in determining which "optional" components are in use in your environment and removing the ones that aren't.

Privilege Elevation via SQL Injection

Most organizations are familiar with the risk posed by SQL injection in web applications, but fewer are aware of the implications of SQL injection in stored procedures. Any component that dynamically creates and executes a SQL query could in theory be subject to SQL injection. In those databases where mechanisms exist to dynamically compose and execute strings, SQL injection in stored procedures can pose a risk.

In Oracle, for example, stored procedures can execute with either the privilege of the invoker of the procedure, or the definer of the procedure. If the definer was a high-privileged account, and the procedure contains a SQL injection flaw, attackers can use the flaw to execute statements at a higher level of privilege than they should be able to. Recently David Litchfield discovered a number of Oracle system–stored procedures that were vulnerable to this flaw (CAN-2004-1370) — the following procedures all allow privilege elevation in one form or another:

DBMS_EXPORT_EXTENSION

WK_ACL.GET_ACL

WK_ACL.STORE_ACL

WK_ADM.COMPLETE_ACL_SNAPSHOT

WK_ACL.DELETE_ACLS_WITH_STATEMENT

DRILOAD.VALIDATE_STMT (independently discovered by Alexander Kornbrust)

The DRILOAD.VALIDATE_STMT procedure is especially interesting since no "SQL injection" is really necessary; the procedure simply executes the specified statement with DBA privileges, and the procedure can be called by anyone, for example the default user "SCOTT" can execute the following:

```
exec CTXSYS.DRILOAD.VALIDATE_STMT('GRANT DBA TO PUBLIC');
```

This will grant the "public" role DBA privileges.

In most other databases the effect of SQL injection in stored procedures is less dramatic — in Sybase, for example, "definer rights" immediately back down to "invoker rights" as soon as a stored procedure attempts to execute a dynamically created SQL statement. The same is true of Microsoft SQL Server.

It isn't true to say that SQL injection in stored procedures has no effect in SQL Server, however — if an attacker can inject SQL into a stored procedure, he can directly modify the system catalog — but only if he already had permissions that would enable him to do so. The additional risk posed by this is slight, since the attacker would already have to be an administrator in order to take advantage of any SQL injection flaw in this way — and if he is a database administrator, there are many other, far more serious things he can do to the system.

One privilege elevation issue in SQL Server is related to the mechanism used to add jobs to be executed by the SQL Server Agent (#NISR15002002B). Essentially, all users were permitted to add jobs, and those jobs would then be executed with the privileges of the SQL Agent itself (by getting the SQL Agent to re-authenticate *after* it had dropped its privileges).

In general, patching is the answer to this class of problem. In the specific case of Oracle, it might be worth investigating which sets of default stored procedures you actually need in your environment and revoking access to "public" — but as we previously noted, this can cause permission problems that are hard to debug.

Local Privilege Elevation Issues

It could be argued that the "unauthenticated access to functionality" class is a subset of this category, though there are some differences. This category is comprised of bugs that allow some level of privilege elevation at the operating system level. Most of the Oracle "extproc" vulnerabilities arguably also fall into this class.

The entire class of privilege elevations from database to operating system users also falls into this class; SQL Server and Sybase's extended stored procedure mechanism (for example, xp_cmdshell, xp_regread), MySQL's UDF mechanism (the subject of the January 2005 Windows MySQL worm), and a recent bug discovered by John Heasman in PostgreSQL (CAN-2005-0227) that allows non-privileged users to load arbitrary libraries (and thereby execute initialization functions in those libraries) with the privileges of the PostgreSQL server.

Other examples of bugs in this category are the SQL Server arbitrary file creation/overwrite (#NISR19002002A), and the SQL Server sp_MScopyscript arbitrary command execution (CAN-2002-0982) issues discovered by David Litchfield.

MySQL had an interesting issue (CAN-2003-0150) in versions prior to 3.23.56, whereby a user could overwrite a configuration file (my.cnf) to change the user that MySQL runs as, thereby elevating MySQL's context to "root." If the user had privileges to read files from within MySQL (file_priv), he would then be able to read any file on the system — and, via the UDF mechanism we discuss later in this volume, execute arbitrary code as "root."

We discuss some recent issues in this category in Informix and DB2 later in this book.

In general, the best defense against this class of bug is to always run your database as a low-privileged user — preferably in a chroot jail, but certainly within a "segregated" part of the file system that only the database can read and write to.

So What Does It All Mean?

The brief summary in the preceding sections has outlined a number of bugs in a small collection of interesting categories, mostly discovered by a small set of people — of which the authors of this volume form a significant (and highly prolific) part. The security research community is growing all the time, but it seems there is still only a small set of individuals routinely discovering security flaws in databases.

What are we to make of this? Does it mean database security is some kind of black art, or that those who are able to discover security bugs in databases are especially skilled? Hardly. We believe that the only reason people haven't discovered more security flaws in databases is simply that people aren't looking.

In terms of the future of database security, this has some interesting implications. If we were being forced to make predictions, our guess would be that an increasing proportion of the security research community will begin to focus on databases in the next couple of years, resulting in a lot more patches — and a lot better knowledge of the *real* level of security of the systems we all depend on so utterly. We're in for an interesting couple of years; if you want to find out more about the security of the systems you deploy in your own network, the next section is for you.

Finding Flaws in Your Database Server

Hopefully the long catalog of issues described in the previous section has you wondering what security problems still lurk undiscovered in your database system. Researching bugs in databases is a fairly convoluted process, mainly because databases themselves are complex systems.

If you want to find security bugs in your database system, there are a few basic principles and techniques that might help:

- Don't believe the documentation
- Implement your own client
- Debug the system to understand how it works
- Identify communication protocols
- Understand arbitrary code execution bugs
- Write your own "fuzzers"

Don't Believe the Documentation

Just because the vendor says that a feature works a particular way doesn't mean it actually does. Investigating the precise mechanism that implements some interesting component of a database will often lead you into areas that are relevant to security. If a security-sensitive component doesn't function as advertised, that's an interesting issue in itself.

Implement Your Own Client

If you restrict yourself to the clients provided by the vendor, you will be subject to the vendor's client-side sanitization of your requests. As a concrete example of this, the overly long username overflow that Mark Litchfield found in Oracle (CAN-2003-0095) was found after using multiple clients, including custom-written ones. The majority of the Oracle-supplied clients would truncate long usernames, or return an error before sending the username to the server. Mark managed to hit on a client that didn't truncate the username, and discovered the bug.

In general, most servers will implement older versions of their network protocols for backward compatibility. Experience tells us that legacy code tends to be less secure than modern code, simply because secure coding has only recently become a serious concern. Older protocol code might pre-date whole classes of security bugs, such as signedness-error-based overflows and format string bugs. Modern clients are unlikely to let you expose these older protocol elements, so (if you have the time) writing your own client is an excellent way of giving these older protocol components a good going-over.

Debug the System to Understand How It Works

The fastest way of getting to know a large, complex application is to "instrument" it — monitor its file system interactions, the network traffic it sends and receives (especially local traffic), take a good look at the shared memory sections that it uses, understand how the various components of the system communicate, and how those communication channels are secured. The Oracle "extproc" library loading issue is an excellent example of a bug that was found simply by observing in detail how the system works.

Identify Communication Protocols

The various components of a database will communicate with each other in a number of different ways — we have already discussed the virtues of implementing your own client. Each network protocol is worth examining, but there are other communication protocols that may not be related to the network that are just as interesting. For instance, the database might implement a file-based protocol between a monitoring component and some log files, or it might store outstanding jobs in some world-writeable directory. Temporary files are another interesting area to examine — several local privilege elevation issues in Oracle and MySQL have related to scripts that made insecure use of temporary files. Broadly speaking, a communication protocol is anything that lets two components of the system communicate. If either of those components can be impersonated, you have a security issue.

Understand Arbitrary Code Execution Bugs

You won't get very far without understanding how arbitrary code execution issues work. Almost everyone is aware of the mechanics of stack overflows, but when you break down arbitrary code execution issues into subcategories, you get interesting families of problems — format string bugs, FormatMessage bugs, sprintf("%s") issues, stack overflows, stack overflows into app data, heap overflows, off-by-one errors, signedness errors, malloc(0) errors — there are a lot of different ways that an attacker can end up running code on the machine, and some of them can be hard to spot if you don't know what you're looking for.

A full description of all of these classes of issues is beyond the scope of this book, however if you're interested, another Wiley publication, *The Shellcoder's Handbook*, might be a useful resource.

Write Your Own "Fuzzers"

Different people have different definitions of the word "fuzzer." Generally, a fuzzer is a program that provides semi-random inputs to some other program and (possibly) monitors the subject program for errors. You could write a fuzzer that created well-formed SQL queries with overly long parameters to standard functions, for example. Or you could write a fuzzer for Oracle TNS commands, or the SQL Server TDS protocol.

When you write a fuzzer, you're effectively automating a whole class of testing. Some would argue that placing your faith in fuzzers is foolish because you lose most of the "feeling" that you get by doing your testing manually. Although a human might notice a slight difference in behavior from one input to the next — say, a brief pause — a fuzzer won't, unless it's been programmed to. Knowledge, understanding, and hard work can't be easily automated — but brute force and ignorance can, and it's often worth doing.

Conclusion

We believe that the best way to secure a system is to understand how to attack it. This concept, while controversial at first sight, has a long history in the field of cryptography and in the broader network security field. Cryptographic systems are generally not considered "secure" until they have been subjected to some degree of public scrutiny over an extended period of time. We see no reason why software in general should not be subject to the same level of scrutiny. Dan Farmer and Wietse Venema's influential 1994 paper "Improving the Security of Your Site by Breaking into It" neatly makes the argument in favor of understanding attack techniques to better defend your network.

This book is largely composed of a lot of very specific details about the security features and flaws in a number of databases, but you should notice common threads running through the text. We hope that by the end of the book you will have a much better understanding of how to attack the seven databases we address directly here, but also a deeper understanding of how to attack databases in general. With luck, this will translate into databases that are configured, maintained, and audited by people who are far more skilled than the people who attack them.