KoreLogic (security) "Securing the most important and sensitive networks and systems in the world."

Home-grown Crypto (Taking a Shiv to a Gun Fight)

March 25, 2007

Hank Leininger <hlein-shmoo2007@korelogic.com>

F980 A584 5175 1996 DD7E C47B 1A71 105C CB44 CBF8

Klayton Monroe <klm-shmoo2007@korelogic.com>

445A CA69 A195 D478 D229 9B89 C0CB C659 6D63 17B4

Cofounders, KoreLogic

©2007 KoreLogic, Inc.™ All Rights Reserved.

10 AM on Sunday is cruel and unusual punishment. Thank you for coming.

©2007 KoreLogic, Inc.™ All Rights Reserved.

- Intro
- Case Studies - Home Grown Crypto Wall of Shame
- Prehash Analysis
- Recommendations and Closing Remarks

©2007 KoreLogic, Inc.™ All Rights Reserved.

- We are not crypto-mathematicians, we're just hackers.
- We regularly see bad crypto.
  - So will you...
- If we can break it, crypto-mathematicians can break it before breakfast.
- We <u>keep</u> seeing these mistakes, so the message bears repeating.

©2007 KoreLogic, Inc.™ All Rights Reserved.

What kinds of things do we see?

- Horrible home-grown “encryption algorithms”
- Giving away the key:
  - For free
  - For a bag of stale Cheetos
  - For a few bucks
- Using good, industry-standard algorithms incorrectly
- Incorrect assumptions about the work factor required to attack the system

©2007 KoreLogic, Inc.™ All Rights Reserved.

- “*Home grown crypto is bad crypto. ... Every programmer tries to build their own encryption algorithm at some point. In one word: Don't.*” Brian Hatch, 2003-01-23
- Security through vigorous handwaving
- Most of the bad crypto we see boils down to security through obscurity, <u>at best</u>

©2007 KoreLogic, Inc.™ All Rights Reserved.

What uses of encryption are we talking about?

- We're not picking on industry-standard algorithms or protocols such as 3DES, AES, or SSL per se ...
- But we will show you cases where we broke implementations protected by them.
- We're talking about using encryption for protecting private data, securing communications, generating session IDs, etc.

©2007 KoreLogic, Inc.™ All Rights Reserved.

- Intro
- Case Studies - Home Grown Crypto Wall of Shame
- Prehash Analysis
- Recommendations and Closing Remarks

©2007 KoreLogic, Inc.™ All Rights Reserved.

- Case Studies - Home Grown Crypto Wall of Shame
  - Obfuscation Gone Bad
  - Keys? We Don't Need No Stinking Keys
  - We Don’t Need Logic, We’ve Got Crypto!
  - Opening Your Neighbour's Garage Door
  - We Have Both Kinds: AES and XOR
  - The House Always… Loses?
  - Can't Crack SSL? Just Talk Plaintext!
  - “Take my data. Please!”

©2007 KoreLogic, Inc.™ All Rights Reserved.

Case Study Format

- Description of system or application tested
- Description of flaw or flaws found
  - May include screen shots, graphics, code, etc.
- How we discovered, broke, or reversed it
- How the flaw could have been prevented and/or how we recommended fixing it
  - But only if “fixing it” was a viable option

©2007 KoreLogic, Inc.™ All Rights Reserved.

Obfuscation Gone Bad

- Online Banking Application

(Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

Obfuscation Gone Bad (1)

- Online Banking Application
  - Awarded a seal of approval from a security company that "certifies" security
  - Encrypted account numbers were used to
    - Access account information
    - Download bank statements and cancelled checks
- Unfortunately, the account numbers weren't really encrypted – they were merely obfuscated with a simple substitution cipher.

©2007 KoreLogic, Inc.™ All Rights Reserved.

Obfuscation Gone Bad (2)

- Notice the FNAME value in the following request:
  - TGXPPGGG0001760000250217XRYYYYY
  - It's a sanitized version of the obfuscated account number ...

```
GET /checkimages/GetImage.asp?FNAME=TGXPPGGG0001760000250
217XRYYYYY&SIDE=FT HTTP/1.0
Referer: https://somebank.owned.com/checkkimages/retrieve
.asp?TGXPPGGG0001760000250217XRYYYYY
Connection: Keep-Alive
Host: somebank.example.com
[snip]
Cookie: ASPSESSIONIDGQQGQQZF=PEIAHJKCGBHNKJIDBEBKNFLG
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

Obfuscation Gone Bad (3)

- The DeEncrypt() routine reveals the ugly truth!
  - Code was obtained from the webserver via a view source bug

```
Sub DeEncrypt(ThisStr,StrLen)
  OldStr = ThisStr
  ThisStr= ""
  For i = 1 to StrLen
    tmpChar = Mid(OldStr,i,1)
    Select Case tmpChar
      Case "G" ThisStr = ThisStr & "0"
      Case "X" ThisStr = ThisStr & "1"
      Case "P" ThisStr = ThisStr & "2"
      Case "K" ThisStr = ThisStr & "3"
      Case "Z" ThisStr = ThisStr & "4"
      Case "L" ThisStr = ThisStr & "5"
      Case "R" ThisStr = ThisStr & "6"
      Case "Y" ThisStr = ThisStr & "7"
      Case "C" ThisStr = ThisStr & "8"
      Case "T" ThisStr = ThisStr & "9"
 [snip]
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

Obfuscation Gone Bad (4)

`TGXPPGGG 0001760000250217 XRYYYYY`

↓ ↓

`90122000 0001760000250217 1677777`

- Is this your account number?
- This scheme made it very easy for us to brute force account numbers and access customer statements and cancelled checks

©2007 KoreLogic, Inc.™ All Rights Reserved.

Keys? We Don't Need No Stinking Keys

- Home-grown Single Sign On (SSO) Application

(Back) (Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

Keys? We Don't Need No Stinking Keys

- Home-grown Single Sign On (SSO) Application
  - SSO between multiple distributed, un-trusting web servers
  - Usernames and passwords were encrypted and sent as cookies to other sites (in the same DNS domain)
  - Each remote site would decrypt the cookie and pre-populate the login form with the decrypted username
  - The initial login was SSL'ed, but the rest of the applications were not, for "performance reasons"

©2007 KoreLogic, Inc.™ All Rights Reserved.

Keys? We Don't Need No Stinking Keys (2)

- The implementation made knowledge of the crypto and encryption keys unnecessary
- Decrypting any encrypted blob was simply a matter of:
  – Constructing a GET request with the target ciphertext added to a special cookie
  – Submit the request
  – Wait for the server's response
  – View the HTML to recover the plaintext

©2007 KoreLogic, Inc.™ All Rights Reserved.

Keys? We Don't Need No Stinking Keys (3)

- A client logs in; POST arguments include:

```
LOGIN_USERID=foo&LOGIN_PASS=q1w2e3r4
```

- The server sends to the client:

```
Set-Cookie:
REMEMBERUSERNAME=d47d9e3234c44c3cbc858b43afb1cfb1
Set-Cookie:
REMEMBERPASSWORD=cb0d2cf4d49d68cf21634f45a7ec7a1a
```

- If we then send the password blob to the server as a username:

```
GET /loginform.jsp HTTP/1.0
Cookie: REMEMBERUSERNAME=cb0d2cf4d49d68cf21634f45a7ec7a1a
```

- The server helpfully decrypts it for us:

```
<input class='bodytxt' type="text"
name="LOGIN_USERID" value='q1w2e3r4'>
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

Keys? We Don't Need No Stinking Keys (4)

- Alone, this flaw was not sufficient to pull off a remote attack (unless you have a nearby victim to sniff). It needed a partner in crime...
- One of the components on one of the SSO’ed webservers was a bulletin-board style app
- Enter XSS – Attack a user via XSS; their encrypted credentials would be exposed to the attacker
- Armed with the ciphertext the attacker could simply issue a modified HTTP request to recover the plaintext

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Don't Need Logic, We've Got Crypto!

- A Web Services application with a Java client

(Back) (Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Don't Need Logic, We've Got Crypto! (1)

- A Web Services application with a Java client
  - All transactions were SOAP messages – HTTPS POSTs of XML data
  - All client messages used SOAP Digital Signatures
  - The server rejected requests without a valid signature
- We thought we were going to need to do some Java disassembly, extract the private key, inject our own, etc.
  - Turned out it was much, much easier than that

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Don’t Need Logic, We’ve Got Crypto! (2)

- A normal client request might look like:

```
POST /services/FooService HTTP/1.0
Content-Type: text/xml; charset=utf-8
...

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope ...>
<soapenv:Header>
<wsse:Security ...>
<wsse:UserIsAdmin>no</wsse:UserIsAdmin>
<ds:SignedInfo>
  ...
</ds:SignedInfo>
<ds:SignatureValue>
pRM0+gKSDkpX5v7mWivz78oWDgvGkiOV8vpDNV1VWlVqz2XP68WpydRalI1IYxWzMwtc
aOMkjJHmnmRyFBAg92QVe7nUoGIxslMgtEzq+8SKyQu7ldF/xFezMQ0fuf9SjuBCHg==
</ds:SignatureValue>
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Don't Need Logic, We've Got Crypto! (3)

- Try to modify the body, and the server will reject it:

```
<?xml version="1.0" encoding="utf-8"?><soapenv:Envelope ...
<soapenv:Body><soapenv:Fault>
<faultcode>soapenv:Server.generalException</faultcode>
<faultstring>WSDoAllReceiver: security processing failed;
nested exception is:
        org.apache.ws.security.WSSecurityException: The
signature verification failed</faultstring>
<detail> ...
```

- We need to extract the private key the client is using, and sign our own messages. Or do we ...... ?

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Don't Need Logic, We've Got Crypto! (4)

- Although the server checked for valid signatures on client inputs, the client neither verified SSL certificates nor expected the server's responses to be signed

- So... we could forge a message from the server to the client using a standard web-proxy, and then the client would spit it back out to the server at the appropriate time, *complete with a valid digital signature*.

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Don’t Need Logic, We’ve Got Crypto! (5)

- It worked. The server checked that the message had a valid signature – and then blindly trusted the contents!
- There was no actual back-end business logic that checked “Hey wait, is user123 really an admin?”
- This is the Web Services version of a webapp that checks what a user is allowed to do when drawing a menu, but doesn't check when an action is actually performed. Only the oldest business-logic flaw in the book.

©2007 KoreLogic, Inc.™ All Rights Reserved.

Opening Your Neighbour's Garage Door

- Enterprise Security Management Software

©2007 KoreLogic, Inc.™ All Rights Reserved.

Opening Your Neighbour's Garage Door (1)

- Enterprise Security Management Software:
  - Centralized Management Server
  - Management Server pushes configs to and harvests logs from end device nodes
  - Fat GUI client application to administer
  - Thin HTTPS-based client to view logs
  - Client-Server and Server-Node communications are encrypted
  - Fat client and thin client use the same username + password credentials

©2007 KoreLogic, Inc.™ All Rights Reserved.

Opening Your Neighbour's Garage Door (2)

- The Server-Node communications uses AES with some kind of feedback (CBC, etc).
- The encryption keys used are user-configurable in the GUI.
- The GUI only allows alphanumeric keys! It also does not enforce any minimum length, or offer to generate strong keys automatically.
- So, AES or not, it uses a user-supplied encryption key with a reduced keyspace.
- Also, the initial block of an encrypted exchange is always the same, suggesting a fixed IV? Maybe we can dictionary-attack it?
- ...But maybe we don't have to...

©2007 KoreLogic, Inc.™ All Rights Reserved.

Opening Your Neighbour's Garage Door (3)

- The Client-Server communications use SSL.
- The Client software has the server's public key bundled with it, so that it can verify the identity of the server.
- The Server software has a private key bundled with it.
- You read that right. *Every installation of this software, worldwide, has the same private key*.
- All you need is a copy of the server software, and you can extract the private key from it.

©2007 KoreLogic, Inc.™ All Rights Reserved.

Opening Your Neighbour's Garage Door (4)

- As it happens, it's possible to download a demo/trial copy.
- The private key is encrypted in the server installation...
- But of course, since the server needs to know how to decrypt it, the passphrase is conveniently sitting in a .properties file in the same directory as the keystore.
- Once decrypted and extracted, use that RSA private key to ssldump management connections to get administrator's passwords. With that, you can see or do anything; reconfigure the entire security architecture if you want.
- Have you ever deployed an enterprise product that uses SSL, and you didn't have to generate certificates for it? This may apply...

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES”

- Network-health monitoring appliance

(Back) (Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (1)

- Network-health monitoring appliance
  - Globally distributed and located in customers' internal networks
  - “Phones home” to appliance vendor to upload data
  - Documentation claimed that sensitive client information was protected through the use of 3DES encryption
- The vendor wanted us to QC their work – kudos to them!
- Unfortunately, the 3DES encrypted keys were being given away

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (2)

- We took a forensic approach:
  - Cracked open the appliance, imaged the hard drive
  - Used FTimes (ftimes.sourceforge.net) to scour the image for clues (credentials, keys, code, DB accounts, etc.)
- A captured transmission revealed that:
  - HTTP version 1.0 and ftp were used to POST data to a vendor site
  - The body of the upload contained binary (3DES?) data that had been encoded as ASCII
- So far so good, right?

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (3)

- We thought so too... until we found the tool that did the encryption:
  - The key was passed in on the command-line
- We traced backwards to find how the key had been generated
- Then, we traced forwards to see how the binary data was generated
- And that's when we saw the key being stuffed in the payload too!

(skip example)

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (4)

A capture of the host reporting home looked random. Well, sort of…

```
POST /cgi-bin/app/app_httpdld HTTP/1.0
Host: upload.example.com
Authorization: Basic YWxseW91cmtleXNhcmU6YmVsb25ndG91cwo=
[snip]
Content-Length: 2441
Content-Type: multipart/form-data

10586010042097632202248O.StatusDownload.rcv
_dftgeqaSay
;<CE[DCDCBFh@h3h>h_^cfd^_^^b`^geda``^``bf^h#D1DEECr?G><?14hDB1>C@?
   BDE[851<D8E[C165\D1Bh?EED213;C5BF5B
8[y@`z6<^g^T`&"XfnEl_U]4>RZ79tL2ei[3Z*nZBz}}FhnAkq$j[_I_\gvzqJ3X)4U0;)
j`axBe8yLCw=xrx"4apa1.;H1VqA]w3lV;"+)+7Yej9c"e_h!]<1:,5a[+FaA;wQC^w{:s
"I;,(:6pQ\LU7+L}2X\R>21i4c8XQh<]0T$_EgzhLYp7Li@R9&z&5R2"3?x8HY2i&#S6R[
A1@=XdsvbBIZRn0/#\X'3wjVn{R5`ABR\Hwg{<W@[hCR<si*b2BQE4@dV[VVNg:*XE2B:4
:$32Y/VNFar)`|G7|/noDr9}5nD]4}
[ snip ]
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

"No Worries, We're Using 3DES" (5)

The encoded stream is also sent via ftp to an upload server.

```
220 FTP server ready.
USER upload2
331 Password required for upload2..
PASS L33tDuD3
230 User upload2 logged in.  Access restrictions apply.
[snip]
STOR 238051003320976321714736.DailyDownload.rcv.
150 Opening BINARY mode data connection for
   238051003320976321714736.DailyDownload.rcv.
`sfrbdgqS_d^y
;<CE[DCDCBFh@hJh4`h`af^c_^^aa`^geda`_e_beadhr19<Ir?G><?14hDB1>C@?BDE[C165\D1Bh?
   EED213;C5BF5B
89UvrdN=Y!I<d9r4a<E]SHinL;_$!pJg[*B(sDU|p,X`vF0}/e_s9fU@b[K{DdW=Iqj1B[/q CU2IR
xQF0Tz|oror2mL[E&if'V)k!4I(|hVyT3NJsH05s"nj2E/i$;o`[`LcSXV1E6/qQy1'S!0V\SvpYI1
(m,SF{Hnq.rqh#)l!k@,$6k> 1;;Yk+T'h3a4nWp6(2\#J35h_C/*x9 )?q[mUweD"$A9?ZlI^#Q,!
IYqZn`3 jW4.|Xe:5G67:z@'.!x|3s0UhK#v!\ ,#=|bINW>d+6"IL2@(c#WQG14Y8h1,z_424`}F.
[snip]
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

"No Worries, We're Using 3DES" (6)

```
#!/usr/bin/perl -wT
use strict;
$::map = <<'EOMAP';
N {" {# {$ {% {& {' {( {) . . . [snip]
EOMAP
@::toenc = split(/ /, $::map);
my @enc_regex;
for (my $i=0; $i<$#::toenc; $i++) {
  $::toasc{ $::toenc[$i] } = $i;
  my $token = $::toenc[$i];
  $token =~ s/([][?{}.+*()&\$|^\\])/\\$1/g;
  push(@enc_regex, $token);
};
$::regex = join('|', @enc_regex);
while(@ARGV and $ARGV[0] =~ /^-([a-zA-Z])/) {
  my $arg = $1;
  die "Usage: $0 [-h] [-x] [-r] [-l string]\n" if ($arg eq 'h');
  $::func = \&Decode if ($arg eq 'r');
  $::hexit = 1 if ($arg eq 'X');
  shift;
};
$::func = \&Encode unless ($::func);
while(<>) {
  &$::func($_);
};
sub Decode {
  while (m/($::regex)/g) {
    print chr($::toasc{$1});
  };
};
sub Encode {
  foreach my $char (split(//,$_)) {
    my $ascii = ord($char);
    my $mapped = $::toenc[$ascii];
[snip]
```

The ASCII encoding wasn't anything standard (uuencode, base64, hex-encoding, etc.). But we had their binary.

After we made a quick dictionary of all possible byte values, a quick perl equivalent encoder / decoder was produced.

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (7)

Decoding the headers provided us with clues.

The three header lines contain transfer metadata: filename, appliance name, transfer type, various control flags, and oh… the 3DES encryption keys.

```
359574702420976322006928.DailyDownload.rcv
tbae_ofeS_`dy
;<CE[DCDCBFh@h3h>hacgcebe^`b`^geda``^^dg`fhr19<Ir?G><?
14hDB1>C@?BDE[C165\D1Bh?EED213;C5BF5B
```

Decoded:

```
359574702420976322006928.DailyDownload.rcv
F4371A87 126K
kls-tstsrv:p:c:n:359574702420976322006928:DailyDownload:
transport-vault.tar:outbackserver
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (8)

Example decoding: putting all the pieces together

Header lines extracted from a transfer:

```
238051003320976321714736.DailyDownload.rcv
`sfrbdgqS_d^y
;<CE[DCDCBFh@hJh4`h`af^c_^^aa`^geda`_e_beadhr19<Ir?G><?
  14hDB1>C@?BDE[C165\D1Bh?EED213;C5BF5B
```

Decoded Header lines:

```
238051003320976321714736.DailyDownload.rcv
2E8D469C 160K
kls-tstsrv:p:z:d2:238051003320976321714736:DailyDownload:
  transport-vault.tar:outbackserver
```

Decode the body using our script:

```
$ encode_char.pl -r <body.tar.enc.hdr.gz.3des.enc
  >body.tar.enc.hdr.gz.3des
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (9)

Example decoding: putting all the pieces together, cont

The key is comprised of the transaction ID + HOSTNAME:
Convert the key to hexadecimal and crop to 32 characters:
The hexadecimal value of the ID always exceeds 32 characters,
making the inclusion of the HOSTNAME in the key irrelevant.

```
echo '238051003320976321714736kls-tstsrv' | encode_char.pl -X | cut -c 1-32
6061665E635F5E5E6161605E67656461
```

Decrypted the data using the key, and ‘des’ from the ancient libdes package:

```
des -d -3 -h -k 6061665E635F5E5E6161605E67656461 body.tar.enc.hdr.gz.3des
    body.tar.enc.hdr.gz
```

Decompress the file:

```
gzip -d body.tar.enc.hdr.gz
```

Remove a redundant header at the end of the file:

```
perl -ne 'print $last; $last=$_' <body.tar.enc.hdr >body.tar.enc
```

Decode the charmapped encoded file:

```
encode_char.pl -r <body.tar.enc >body.tar
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (10)

- Here's a sample payload after it had been cracked

```
tar -tvf body.tar
-rw-rw-r-- tuser/users       0 2004-05-14 15:08 out/logs.download
-rw-rw-r-- tuser/users       0 2004-05-14 15:08 out/tuserreportrundetailsvault.download
-rw-rw-r-- tuser/users       0 2004-05-14 15:08 out/tuserbxmcardresourcevault.download
-rw-rw-r-- tuser/users       0 2004-05-14 15:08 out/tuserbxmchannelusagevault.download
-rw-rw-r-- tuser/users     571 2004-05-14 15:08 out/modulesvault.download
-rw-rw-r-- tuser/users     924 2004-05-14 15:08 out/modulesattrvault.download
-rw-rw-r-- tuser/users     143 2004-05-14 15:08 out/downloadmastervault.download
-rw-rw-r-- tuser/users    5357 2004-05-14 15:08 out/schedulervault.download
-rw-rw-r-- tuser/users       0 2004-05-14 15:08 out/dailyrvault.download
-rw-r----- tuser/users  187204 2004-05-14 15:08 out/logs/seedfmgr.log
-rw-r----- tuser/users 1047963 2004-05-14 15:08 out/logs/scheduler.log
[snip]
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

“No Worries, We're Using 3DES” (11)

- The information was encrypted with 3DES prior to transfer (as claimed in the documentation)
- It was then ASCII encoded with a substitution cipher
- BUT... the first few bytes of the transfer contained the encryption key used to encrypt the rest of the file!
- The vendor didn’t mention that... :/

(skip back)

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR

- “Convergent technologies” application

(Back) (Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (1)

- "Convergent technologies" application:
  - Fat MS Windows client
  - Clients for both x86 W2K, and WinCE
  - Mixture of web-based and proprietary communications
  - AES encryption used in some places and XOR encoding used in others
- Both had problems...

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (2)

- Initially, we found the secret AES key in a DLL using traditional reversing techniques
  - Strings, hex dumps, static disassembly of subtle routines like: Encrypt() and Decrypt()
  - Umm, great key guys: “1234567890abcdef”
- Later, we used FTimes in dig mode to search all files for the observed key pattern:
  - DigStringRegExp=\x00[0-9A-Fa-f]{16}\x00
  - This often reveals additional keys, but not this time

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (3)

- Their choice of key suggested that the total key space was severely limited:
  - Total key space $=256^{16}$ or 2^{128}
    - 340,282,366,920,938,463,463,374,607,431,768,211,456
  - Hex characters only key space = 16^{16} or 2^{64}
    - 18,446,744,073,709,551,616
- The difference is huge!
- But is the crippled key space big enough?
  - Are there other limiters in play?

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (4)

- Initially, our plan was to use the recovered key and an alternate AES implementation to handle our encryption/decryption needs
- But the exported encrypt/decrypt routines were too enticing, so we chose to abuse them instead
  - Eliminated potential implementation gotchas
  - Required disassembly to determine call setup
  - But the results were worth the effort...

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (5)

- Here's a pseudo C implementation that shows how we used the exported Decrypt routine

```
...
#define KEY "1234567890abcdef"
typedef DWORD(__stdcall *MYROUTINE)(LPTSTR,LPTSTR,DWORD,LPTSTR);
int main() {
  ...
  Handle = LoadLibrary("Application.dll");
  if (Handle != NULL) {
    Routine = (MYROUTINE) GetProcAddress(Handle, "Decrypt");
    if (Routine != NULL) {
      Routine(CipherText, PlainText, CHUNK_SIZE, Key);
...
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (6)

- Now we could encrypt and/or decrypt
  - Local files (e.g., user profile)
  - Messages transmitted to/from the server
- This allowed us to craft custom files and messages
- It also revealed a new problem:
  - The implementation used ECB instead of CBC
    - ECB – Electronic Code Book
    - CBC – Cipher Block Chaining
  - ECB was not a good choice in this case... (skip example)

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (7)

```
00000000 c5 e1 c6 6f 93 4a 73 f1 5a 49 e5 27 fe d9 b6 58 |...o.Js.ZI.'...X|
00000010 e3 6f 87 24 f3 b1 ed XX XX XX XX XX XX XX XX XX |.o.$............|
00000020 ef 0b f2 b7 c9 0f b5 e9 80 de 51 9a 16 f8 82 40 |...........Q....@|
00000030 f0 60 a8 d5 35 74 84 1c e5 97 43 XX XX XX XX XX |.`..5t....C.....|
00000040 ff 5b 0c e9 07 8b d0 8e d2 fe a4 0e fb a5 ca 5d |.[.............]|
```

```
00000000 3c 43 75 73 74 6f 6d 65 72 3e 3c 41 63 63 6f 75 |<Customer><Accou|
00000010 6e 74 4e 61 6d 65 3e XX XX XX XX XX XX XX XX XX |ntName>XXXXXXXXX|
00000020 34 3c 2f 41 63 63 6f 75 6e 74 4e 61 6d 65 3e 3c |4</AccountName><|
00000030 47 72 6f 75 70 73 3e 3c 49 44 3e XX XX XX XX XX |Groups><ID>XXXXX|
00000040 3c 2f 49 44 3e 3c 4e 61 6d 65 3e 46 72 69 65 6e |</ID><Name>Frien|
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (8)

```
00000000 c4 e1 c6 6f 93 4a 73 f1 5a 49 e5 27 fe d9 b6 58 |...o.Js.ZI.'...X|
00000010 e3 6f 87 24 f3 b1 ed XX XX XX XX XX XX XX XX XX |.o.$............|
00000020 ef 0b f2 b7 c9 0f b5 e9 80 de 51 9a 16 f8 82 40 |..........Q....@|
00000030 f0 60 a8 d5 35 74 84 1c e5 97 43 XX XX XX XX XX |.`..5t....C.....|
00000040 ff 5b 0c e9 07 8b d0 8e d2 fe a4 0e fb a5 ca 5d |.[.............]|
```

```
00000000 71 f7 95 1e 19 d3 bd 5e 6d 71 e9 eb 45 18 61 84 |q......^mq..E.a.|
00000010 6e 74 4e 61 6d 65 3e XX XX XX XX XX XX XX XX XX |ntName>XXXXXXXXX|
00000020 34 3c 2f 41 63 63 6f 75 6e 74 4e 61 6d 65 3e 3c |4</AccountName><|
00000030 47 72 6f 75 70 73 3e 3c 49 44 3e XX XX XX XX XX |Groups><ID>XXXXX|
00000040 3c 2f 49 44 3e 3c 4e 61 6d 65 3e 46 72 69 65 6e |</ID><Name>Frien|
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (9)

```
00000000 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
00000010 e3 6f 87 24 f3 b1 ed XX XX XX XX XX XX XX XX XX |.o.$............|
00000020 ef 0b f2 b7 c9 0f b5 e9 80 de 51 9a 16 f8 82 40 |..........Q....@|
00000030 f0 60 a8 d5 35 74 84 1c e5 97 43 XX XX XX XX XX |.`..5t....C.....|
00000040 ff 5b 0c e9 07 8b d0 8e d2 fe a4 0e fb a5 ca 5d |.[.............]|
```

```
00000000 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
00000010 6e 74 4e 61 6d 65 3e XX XX XX XX XX XX XX XX XX |ntName>XXXXXXXXX|
00000020 34 3c 2f 41 63 63 6f 75 6e 74 4e 61 6d 65 3e 3c |4</AccountName><|
00000030 47 72 6f 75 70 73 3e 3c 49 44 3e XX XX XX XX XX |Groups><ID>XXXXX|
00000040 3c 2f 49 44 3e 3c 4e 61 6d 65 3e 46 72 69 65 6e |</ID><Name>Frien|
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (10)

```
00000000 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
00000010 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
00000020 ef 0b f2 b7 c9 0f b5 e9 80 de 51 9a 16 f8 82 40 |..........Q....@|
00000030 f0 60 a8 d5 35 74 84 1c e5 97 43 XX XX XX XX XX |.`..5t....C.....|
00000040 ff 5b 0c e9 07 8b d0 8e d2 fe a4 0e fb a5 ca 5d |.[.............]|
```

```
00000000 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
00000010 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
00000020 34 3c 2f 41 63 63 6f 75 6e 74 4e 61 6d 65 3e 3c |4</AccountName><|
00000030 47 72 6f 75 70 73 3e 3c 49 44 3e XX XX XX XX XX |Groups><ID>XXXXX|
00000040 3c 2f 49 44 3e 3c 4e 61 6d 65 3e 46 72 69 65 6e |</ID><Name>Frien|
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (11)

```
00000000 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
00000010 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
00000020 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
00000030 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
00000040 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 |................|
```

```
00000000 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
00000010 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
00000020 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
00000030 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
00000040 a4 07 9e f2 f5 57 6c 70 88 83 25 f0 86 11 8a e5 |.....Wlp..%.....|
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (12)

- Did we mention that the DLL in question was deployed on both client and server components?
- This implied that any vulnerabilities in the DLL could potentially be leveraged on the server
  - Buffer overflows in the client-server traffic
  - Also, this gives an attacker an advantage since s/he has plenty of time to analyze the DLL offline

(Skip back)

©2007 KoreLogic, Inc.™ All Rights Reserved.

We Have Both Kinds: AES and XOR (13)

- And now, the other problem: XOR
- The application allowed user credentials to be “remembered”
  - Stored in the Registry
  - Obscured with a position based XOR
  - Trivial to reverse
- What were they thinking?
- We've been down that road before... <sigh>

©2007 KoreLogic, Inc.™ All Rights Reserved.

The House Always… Loses?

- Online gambling application

(Back) (Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

The House Always… Loses? (1)

- Online gambling application
  - Fat MS Windows client
  - Used a combination of web-based and home-grown protocols
  - Login credentials were hashed/obfuscated and stored in a local INF file
- Unfortunately, the credentials were not completely obfuscated and the scheme was cracked

©2007 KoreLogic, Inc.™ All Rights Reserved.

The House Always… Loses? (2)

- It began with the username, which looked like a SHA1 (or any 160-bit) hash that had been specially formatted:
  - 35675E68-F4B5AF7B-995D9205-AD0FC438-42F16450
  - ^([0-9A-F]{8}-){4}[0-9A-F]{8}
- A simple test (guest/guest) confirmed our theory:
  - echo -n "guest" | openssl sha1 | perl -p -e 's/(.{8})/uc($1)."-"/ge; s/-$//;'
- Using that as our basis, we conjectured that password obfuscation would follow a similar pattern

©2007 KoreLogic, Inc.™ All Rights Reserved.

The House Always… Loses? (3)

- The obfuscated password hash looked like a SHA1 too, but unlike the username hash, it was Base64 encoded:
  - VcqTUWCKPxpAhH8HBIL31kLxZFA=
  - ^[+/0-9A-Za-z]{27}=
- This time, a simple test <u>did not</u> confirm our theory:
  - echo -n "guest" | openssl sha1 -binary | openssl base64
  - NwdeaPS1r3uZXZIFrQ/EOELxZFA=
- However, it revealed something very interesting...

©2007 KoreLogic, Inc.™ All Rights Reserved.

The House Always… Loses? (4)

- The low 32 bits of the obfuscated password hash were identical to the low 32 bits of the password's SHA1 hash
  - bdca9351608a3f1a40847f070482f7d642f16450
  - 35675e68f4b5af7b995d9205ad0fc43842f16450
- The obfuscation algorithms were recovered after several hours of disassembly
- From our perspective, it looked like they had upgraded from 128-bit to 160-bit hashes at some point (and shortcuts were taken)

©2007 KoreLogic, Inc.™ All Rights Reserved.

The House Always… Loses? (5)

- Obfuscation routines were ported to C, which allowed us to produce our own credentials externally and much faster than going through the application
- Password cracking could have been done even without the disassembly effort
- Based on the amount of assembly analyzed, we conjecture that a significant amount of work went into obfuscating credentials – perhaps more than it took to de-obfuscate them

©2007 KoreLogic, Inc.™ All Rights Reserved.

Can't Crack SSL? Just Talk Plaintext!

- A mobile application

(Back) (Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

Can't Crack SSL? Just Talk Plaintext! (1)

- A mobile application
  - Fat BREW client that communicates via HTTPS
  - Strong/Encrypted authentication discouraged MITM attacks and eliminated visibility between handsets and servers
  - The provided handset was in test mode, so the application could be tweaked
    - NOTE: BREW applications are digitally signed, so this line of attack is normally cut off
- Suppose BREW protections <u>could</u> be bypassed...

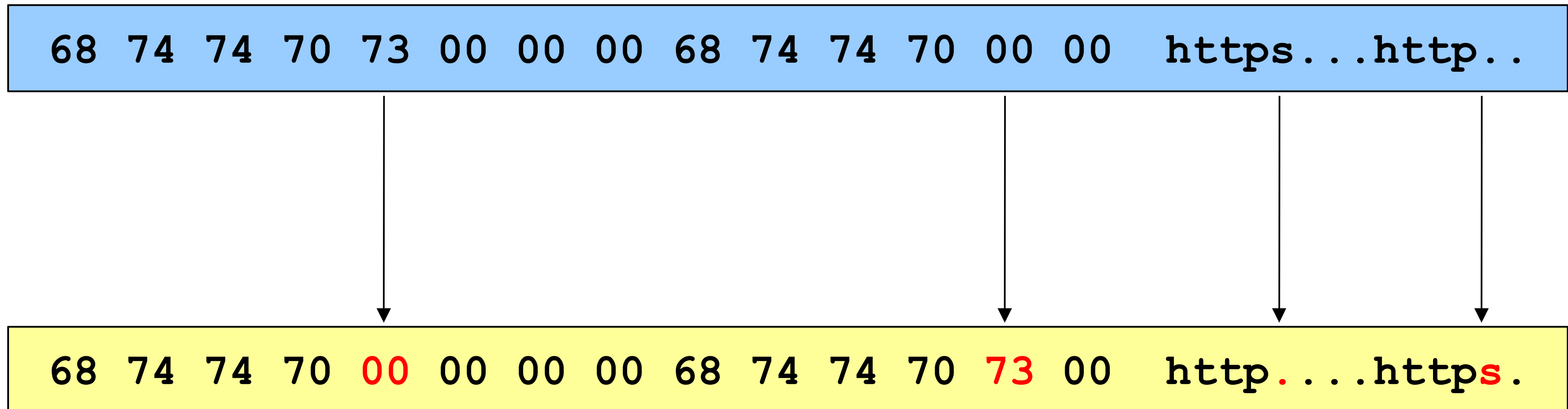
©2007 KoreLogic, Inc.™ All Rights Reserved.

Can't Crack SSL? Just Talk Plaintext! (2)

- We decided to find out how the system would react if the binary was modified to speak HTTP rather than HTTPS
  - Trivial binary edit – no assembly required
- The results were interesting
  - The handset happily spoke HTTP
  - The servers accepted the unencrypted traffic, and, most importantly, certificate authentication was gone!

©2007 KoreLogic, Inc.™ All Rights Reserved.

Can't Crack SSL? Just Talk Plaintext! (3)



- The modification was a simple flip-flop of strings
- With the cloak removed, we were free to study and attack client-server communications at will
- MITM attacks were back on the menu simply by modifying URL addresses

©2007 KoreLogic, Inc.™ All Rights Reserved.

Can't Crack SSL? Just Talk Plaintext! (4)

- The primary flaw, in our opinion, was the belief that the devices couldn’t be hacked
- Designers should assume that all components of a system will be known by or become visible to the attacker sooner or later
  - We found leaked copies of the device flash images, and of tools that would put the devices into test mode, on public websites
- Designers should use every layer of protection to their advantage, but they should not assume that any one layer will suffice
  - Try to design each layer as if it were the <u>only</u> layer

©2007 KoreLogic, Inc.™ All Rights Reserved.

Can't Crack SSL? Just Talk Plaintext! (5)

- If a production system doesn't require unencrypted communications, don't deploy production binaries and services that support unencrypted communications

- If a mixed HTTP/HTTPS webserver starts getting queries in plaintext that ought to be SSL-only… maybe that's a clue!

©2007 KoreLogic, Inc.™ All Rights Reserved.

“Take my data. Please!”

- A compartmentalized manufacturing application

(Back) (Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

“Take my data. Please!” (1)

- A compartmentalized manufacturing application
  - Fat MS Windows client
  - Proxy account built into the client issues an SQL query to authenticate the user's login credentials and obtain the user's encrypted database credentials
  - Database traffic was not encrypted, but some of the individual fields were
- Unfortunately, this application used hard-coded keys, weak obfuscation techniques, and it leaked other users' encrypted credentials!

©2007 KoreLogic, Inc.™ All Rights Reserved.

"Take my data. Please!" (2)

- The authentication process started like this:
  - User launches application (FOO.EXE)
  - User populates login form (user123/hammer)
  - Application issues the query shown below using hard-coded database credentials
    - u1, u2 = encrypted login username/password
    - u3, u4 = encrypted database username/password

SELECT id, u1, u2, u3, u4 FROM common_table WHERE somekey = 22;

©2007 KoreLogic, Inc.™ All Rights Reserved.

“Take my data. Please!” (3)

- Next, the authentication process did this:
  - Decrypt u1 of each record until a match is found for the current user (user123)
  - Decrypt u2 to determine if password matches the one supplied by the current user (hammer)
  - If both u1 and u2 checks pass, decrypt u3 and u4 to obtain database credentials
  - At this point, the user could work on the project
- *All of these checks are done locally, in the client!*

©2007 KoreLogic, Inc.™ All Rights Reserved.

"Take my data. Please!" (4)

- We could sniff the results of the query and could tell that the fields were encrypted
- We used OllyDbg to help us analyze the application
  - The encryption/decryption key was stored in a local file: "query.key"
    - 1536 bytes of binary data
  - After some custom code, DLL abuse, and disassembly, we found the key, but it was only 57 bytes long
  - So why was "query.key" so much bigger?

©2007 KoreLogic, Inc.™ All Rights Reserved.

"Take my data. Please!" (5)

OllyDbg - d.exe - [CPU - main thread, module crypto]

KEY

Breakpoint at crypto.10001D6A

Paused

©2007 KoreLogic, Inc.™ All Rights Reserved.

"Take my data. Please!" (6)

- We recovered the algorithm and wrote an extraction program to pull the key...

```
...
#define INITIAL_OFFSET 0x232
#define STEP 7
#define SIZE 1536
#define KEY_LENGTH 0x39
int main() {
  ...
  iNRead = fread(pcData, 1, SIZE, pFile);
  ...
  for (i = 0, iOffset = INITIAL_OFFSET; i < KEY_LENGTH; i++, iOffset += STEP) {
    fprintf(stdout, "%02x,", pcData[iOffset]);
  }
  fprintf(stdout, "\n");
  return 0;
}
```

©2007 KoreLogic, Inc.™ All Rights Reserved.

“Take my data. Please!” (7)

- Because multiple records were returned in the initial SQL query, we were able to decrypt other user's credentials
  - In fact, we were able to modify the query to select ALL records!
- We wrote a C program to abuse the exported DLL routines and the recovered key
  - LoadLibrary(), GetProcAddress(), and a bit of inline assembly
- Sure, we'll take your data... :)

©2007 KoreLogic, Inc.™ All Rights Reserved.

- Case Studies - Home Grown Crypto Wall of Shame
  - Obfuscation Gone Bad
  - Keys? We Don't Need No Stinking Keys
  - We Don’t Need Logic, We’ve Got Crypto!
  - Opening Your Neighbour's Garage Door
  - We Have Both Kinds: AES and XOR
  - The House Always… Loses?
  - Can't Crack SSL? Just Talk Plaintext!
  - “Take my data. Please!”

©2007 KoreLogic, Inc.™ All Rights Reserved.

- Intro
- Case Studies - Home Grown Crypto Wall of Shame
- Prehash Analysis
- Recommendations and Closing Remarks

(Skip)

©2007 KoreLogic, Inc.™ All Rights Reserved.

Prehash Analysis

- Session IDs are often created by combining several of the following variables:
  - Username, realm, IP address, device ID, resource, time, shared secrets, nonces, etc.
- Typically, several variables are concatenated and hashed with an algorithm such as MD5 or SHA1
- Too few, easily guessed, or badly ordered variables are common problems
  - Application context often makes all the difference

©2007 KoreLogic, Inc.™ All Rights Reserved.

Prehash Analysis (2)

- Sometimes, an appropriate number of variables are chosen, but they're ordered in a way that makes the scheme weak
- Our MD5 prehash analysis confirmed that weak schemes could be attacked faster
- The idea was to find a way to decrease the amount of time required to brute force a particular scheme
- In particular, we focused on a class of IDs that combine two types of variables: fixed prefix and variable suffix

©2007 KoreLogic, Inc.™ All Rights Reserved.

Prehash Analysis (3)

- Fixed Prefix = Easy to guess or infer
- Variable Suffix = Hard to guess or random
- Prehashing the fixed portion allows overall hashes to be computed faster
- If the variable portion is too short, this could be a real problem

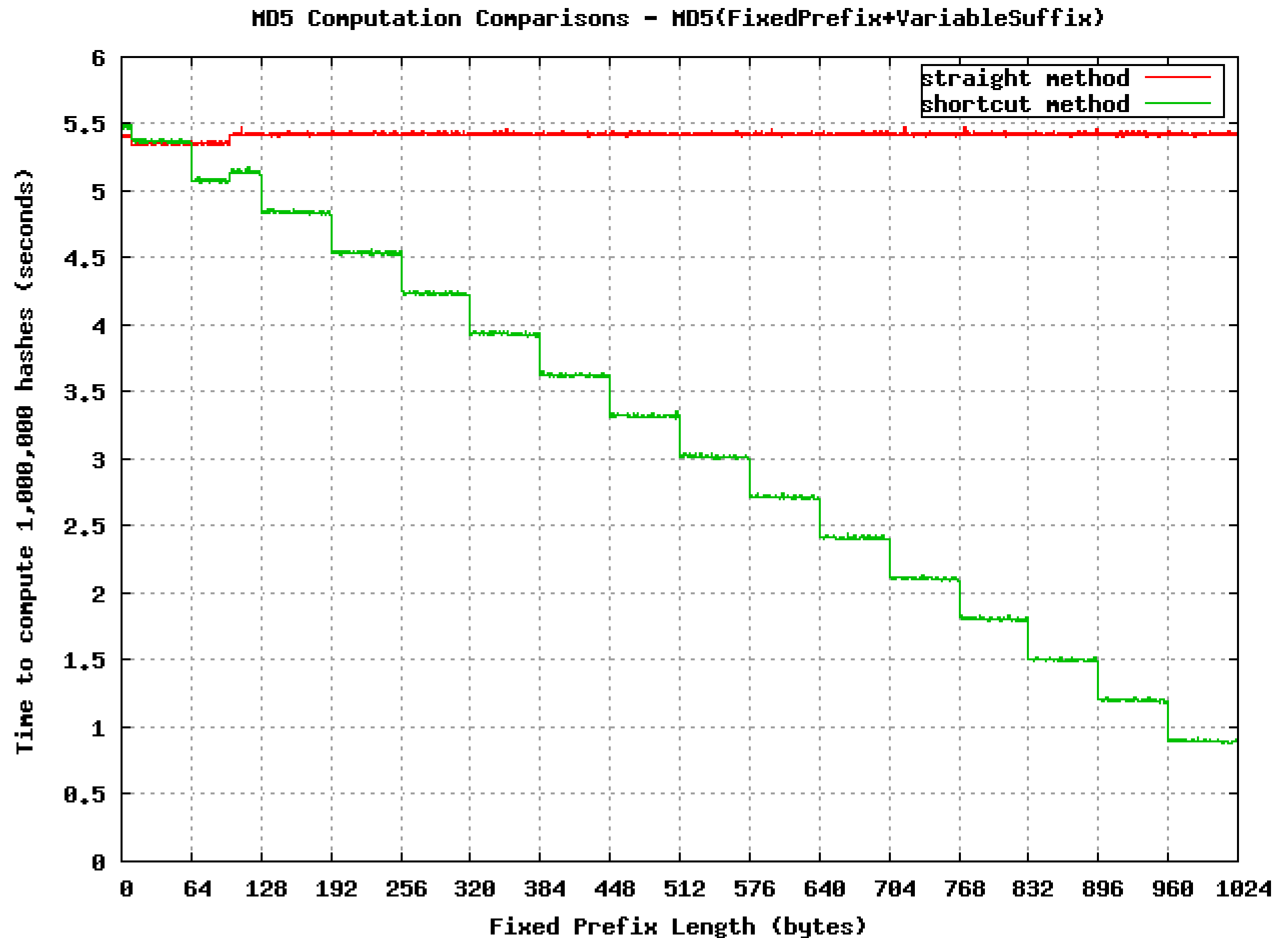
©2007 KoreLogic, Inc.™ All Rights Reserved.

Prehash Analysis (4)

- There is a significant advantage to prehashing if the hard-to-guess variable(s) are at the end of the string and the total string length is longer than 64 bytes
- At 512 bytes, the performance gain is roughly 2:1; at 1024, bytes the gain is roughly 6:1 (see graph)
- The results of our analysis are documented here:
  - www.korelogic.com/Resources/Papers/md5-analysis-prehash.txt
- Rainbow-table techniques should be applicable in some cases, for the same reason, making fixed-prefixes even worse

©2007 KoreLogic, Inc.™ All Rights Reserved.

Prehash Analysis (5)



©2007 KoreLogic, Inc.™ All Rights Reserved.

Prehash Analysis (6)

- Session ID: 1020-byte fixed prefix, 4-byte variable suffix
  - Prehash method will crack the ID in ~1 hour
  - Regular method will crack the ID in ~6.5 hours
  - Time required to try each ID is not included here
- Put the hard-to-guess stuff at or near the front
  - hash(unknown,known,known) – DO
  - hash(known,known,unknown) – DON'T
- Assume the easy-to-guess stuff will be lost
  - context, loose lips, lucky/educated guesses, etc.

©2007 KoreLogic, Inc.™ All Rights Reserved.

Prehash Analysis (7)

- Don't make the hard-to-guess stuff too short
  - We've seen IP addresses, SSNs, timestamps, PIDs, phone numbers, etc. used as the “hard-to-guess” stuff
  - Too often, the space for these items can be reduced:
    - Targeting phone numbers in particular area code and exchange reduces the space from 10^{10} to 10^4
    - Targeting SSNs in Maryland (212-220) reduces the space from 10^9 to less than 10^7 (9×10^6); targeting specific years of birth can reduce it still further
    - Targeting IPs can be dead simple if the users all have a common ISP!
- Include brute force detection and logging in your designs

©2007 KoreLogic, Inc.™ All Rights Reserved.

- Intro
- Case Studies - Home Grown Crypto Wall of Shame
- Prehash Analysis
- Recommendations and Closing Remarks

©2007 KoreLogic, Inc.™ All Rights Reserved.

Don't Grow Your Own Crypto

- Unless you are a crypto mathematician and your job is to build crypto algorithms
- There are many ways to get it wrong...
- New algorithms require years of analysis by many experts to be verified and accepted
- Proprietary algorithms won't get the level scrutiny that open algorithms will
- Just because you can't break it, doesn't mean that someone else can't break it

©2007 KoreLogic, Inc.™ All Rights Reserved.

Don't Assume Too Much

- “That'll never happen...” (epidemic)
- Plan on multiple protection mechanisms failing or being bypassed rather than assuming they'll just work
- “Don't underestimate the power of the dark side...”
  - Software disassembly techniques and tools are getting better all the time
  - If your secrets are important enough, someone will attempt to get them

©2007 KoreLogic, Inc.™ All Rights Reserved.

Don't Rely On Obfuscation

- It's very difficult to prevent algorithm recovery when the attacker has full access to the client software, device, or whatever...
- Your time is better spent hardening server-side components and placing zero trust in client-side components that are out of your control
- Assume that any obfuscation you use is transparent
- Don't hide keys where they can be recovered – even if you think they can't be recovered

©2007 KoreLogic, Inc.™ All Rights Reserved.

Design From Your Data Out

- Design security around your data and work outwards from there – not the other way 'round
- Use layers of protection whenever possible
  - Assume that each layer will be broken
  - Add detection and logging at every layer
- Keep protection mechanisms simple and modular
- Compartmentalize your data
  - Design such that a breach of one compartment does not automatically imply a breach of all

©2007 KoreLogic, Inc.™ All Rights Reserved.

Design For Agility

- Crypto Agility
  - Hashes, keys, key lengths, algorithms, and RNGs
- Protocol Agility
  - How will your application handle protocol flaws?
- Error Agility
  - Does your application handle unexpected errors gracefully?
- Application Agility
  - How do you plan to upgrade all those clients?

©2007 KoreLogic, Inc.™ All Rights Reserved.

Use Care When Implementing Crypto

- Know how and why you are using it
- Use standard libraries that have already worked out most of the kinks
- Use previously stated guiding principles
  - Don't grow your own crypto
  - Don't assume too much
  - Don't rely on obfuscation
  - Design from your data out
  - Design for agility

©2007 KoreLogic, Inc.™ All Rights Reserved.

Hardware Trumps Software

- Generally, it costs more to reverse crypto and/or protection mechanisms implemented in hardware (if done well)
  - NOTE: We're not talking about appliances built around general purpose computers
- But don't get complacent... Hardware reversing is just a matter of time and money
  - What is the life expectancy of your product (e.g., client software, mobile devices, appliances)?
  - How much time and money do you want the attackers to spend attacking your product?

©2007 KoreLogic, Inc.™ All Rights Reserved.

That's All Folks... Thanks!

- www.korelogic.com

- Random plug: http://marc.info

- Extra credit: Why did the fly dance on the jar?

©2007 KoreLogic, Inc.™ All Rights Reserved.

This space intentionally left blank.

©2007 KoreLogic, Inc.™ All Rights Reserved.

This space is even more intentionally left blank. (We miss you, Infocom.)

©2007 KoreLogic, Inc.™ All Rights Reserved.