Web App Cryptology

A Study in Failure

Travis H.

OWASP AppSec USA 26 Oct 2012

Bay Area Hacker's Association

- Meets once a month
- `http://baha.bitrot.info/`

Why Study Failure?

Quotes

"Few false ideas have more firmly gripped the minds of so many intelligent men than the one that, if they just tried, they could invent a cipher that no one could break."
– David Kahn

"Those who cannot learn from history are doomed to repeat it."
– George Santayana

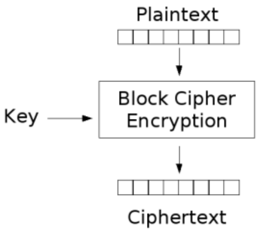
- Nobody *really* knows how to make unbreakable crypto, so learn how to make things that aren't breakable by any known technique, and hope for best

Where Crypto Is Needed in Web Apps

- Hidden Fields
- GET parameters
- POST parameters
- Cookies (especially *authenticators*, see next slide)
- Anything that gets sent to clients and intended to be returned unaltered

Authenticators

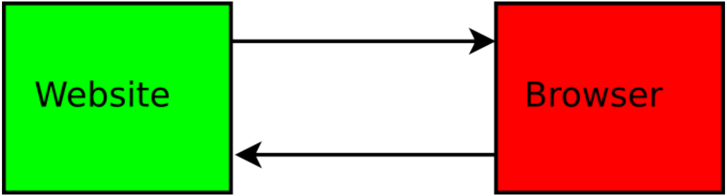
- Indicate that user has gone through login process
- Used instead of HTTP auth
- Implies or includes login name (usually)
- Can't be stored plaintext, so typically encrypted: $C = E_K(P)$
- C is ciphertext (stored in cookie), K is key, P is plaintext (identifier)



Normal Encryption

- Sender sends message through Internet to recipient
- Large number of sender/recipient pairs suggests PK

Your Problem



- Sending data to yourself through the browser

About Unix crypt(3)

- Library function used for hashing system passwords; ***not encryption routine!***
- Is really close to DES encryption of a *plaintext* of all-zeroes using the input as the *key*
- Inputs reversed from most *encryption* routines
- Depends on being unable to determine the *key* given the *ciphertext*

Crypting with Salt

- 12 bits of "salt" used to perturb the encryption algorithm, so off-the-shelf DES hardware implementations can't be used to brute-force it faster
- Salt should be random, else identical passwords hash to identical values
- Salt and the final ciphertext are encoded into a printable string in a form of base64

How Unix crypt(3) Works

1. User's password truncated to 8 characters, and those are coerced down to only 7 bits ea.
2. Forms 56-bit DES key
3. Salt used to make part of the encryption routine different
4. That is then used to encrypt an all-bits-zero block: $crypt(s, x) = s + E_x(\bar{0})$
5. Iterate 24 more times, each time encrypting the results from the last round
6. Repeating makes it slower (on purpose)

WSJ.com Flaw #1

WSJ Authenticator

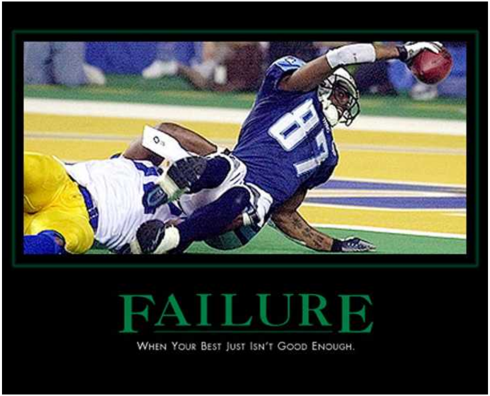
- let + be concatenation
- Unix crypt (salt, username + secret string)
- = salt + encrypted_data
- = WSJ.com authenticator

- Hint: Where is the secret string located?

WSJ.com Long Name Instant Fail

- Unix crypt(3) only hashes 8 octets, so truncates input string
- $crypt(s, "dandylionSECRETWORD") \equiv crypt(s, "dandylio")$
- Pick an 8 character username
- Pick a salt
- Do the crypt yourself
- Presto: you have a valid authenticator for that username w/o knowing secret string

WSJ Failure #1



- crypt(3) is not a encryption routine
- wrong tool for the job

WSJ.com Salt Failure

- Usernames identical in the first 8 letters had identical authenticators
- Thus interrogative adversary can observe salt was fixed constant in the program
- Means that I can use one authenticator with another user's login
- Assuming both usernames start with same 8 characters

WSJ.com Failure #2

- No two authenticators should be the same
- LOL WTF R U DOING?

WSJ.com Flaw #3

WSJ Authenticator

- crypt (salt, username + secret string)
- = salt + encrypted_data
- = the WSJ.com authenticator

Hint: This problem allows you to recover the secret string easily

Adaptive Chosen Message Attack

WSJ Authenticator

- crypt (salt, username + secret string)

1. Register username "failfai"
2. compute $crypt(s, \text{"}failfaiA\text{"})$ and see if that's a valid authenticator for user failfai
3. If not, pick a different letter and try step 2 again.
4. If it is, you know first letter of secret string.
5. Reduce username length by one, register it and jump to step 2
6. When this stops working you've gotten all of the key

WSJ Flaw #3

- By adaptive chosen message attack, can be broken in $128x8$ iterations instead of 128^8
- Each query took 1 second
- Secret string was "March20"

Time is O(n) instead of O(c^n)

- ACMA gives full key recovery in 17 minutes
- ...Instead of $2x10^9$years

WSJ Epic Fail

- 17 minutes to recover "secret"
- ancient analytic technique going back to TENEX systems

Poor Random Number Generation

- The best crypto can't save you from a broken RNG
- Netscape SSL flaw (1995)
- MS CryptGenRandom (Nov 2007)
- Dual_EC_DRBG (Aug 2007)
- Debian OpenSSL (May 2008)

Hashes Generally

- *Cryptographic* hashes are one way functions
- Given input, it's easy to compute output
- Given the output, it's difficult to compute input
- Tiny change in input = big change in output

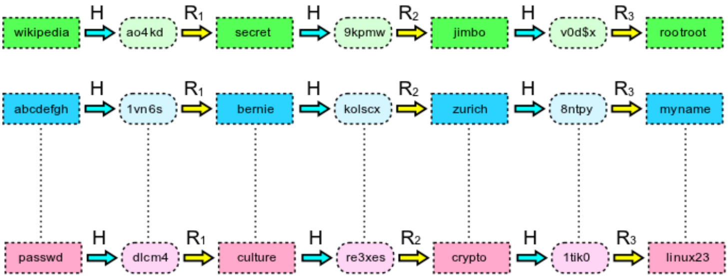
Hashing With No Salt

- Allow user to pick secret s - easy to guess
- Don't want to store user secrets in plaintext form
- Pass through a (crypto) hash instead, store digest
- Any guesses what is wrong with this?

Hashing With No Salt Flaw

- Simply hash all likely secrets
- Already done in rainbow tables you can download

Rainbow Tables



- Essentially a clever way to store precomputed hashes
- Easy to download for most hashes over alphanumerics
- Can easily look up any unsalted precomputed hash

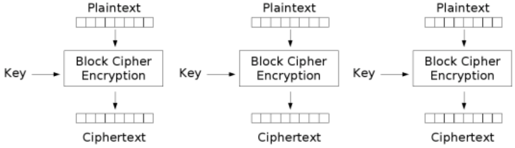
Hashing With Salt

- Whenever you're hashing weak (easy to guess) secrets
- Always prepend a unique, random byte series to the secret and the hash output
- $salthash(s, i) = s + hash(s + i)$
- I recommend using as many bits of salt as your hash has output
- This guarantees rainbow tables would have to hash every input, not just likely inputs

Password Hashing Alternatives

- Use HMAC (described later) instead of simple hash, with salt as the key
- Better yet, use PBKDF2 for passwords. This iterates 1000 times (recommended minimum) on each password, making cracking passwords much more time consuming.

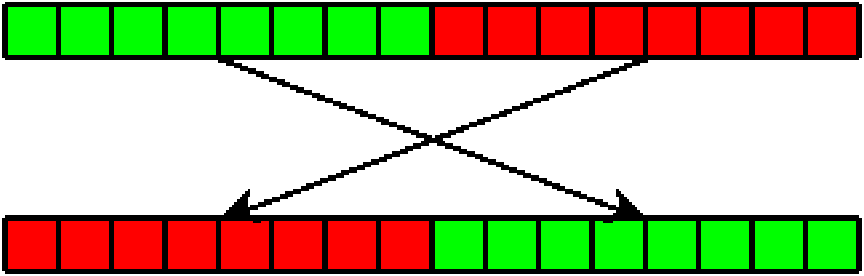
What Is ECB Mode?



Electronic Codebook (ECB) mode encryption

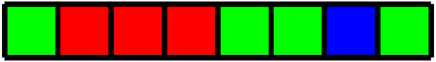
$C_i = E_K(P_i)$ done independently for each *block* of plaintext

ECB Block Swapping



- Adversary can swap ciphertext *blocks* around and effectively swap plaintext *blocks* around without breaking crypto
- AAAAAAAABBBBBBBB can be changed to BBBBBBBBAAAAAAAA

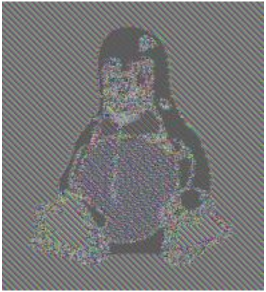
ECB Block Repetition

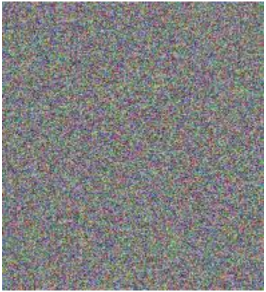


- Any plaintext *block* that repeats later in the stream will show repetition in the ciphertext
- The *blocks* above show a pattern of ABBBAACA
- Fails to destroy macroscopic patterns in the plaintext; any pattern that is present above the block level remains a pattern in the ciphertext.

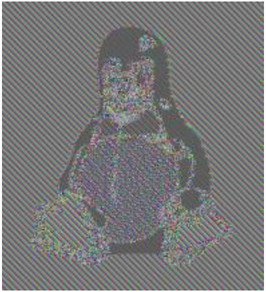
Using ECB Mode


plaintext


ECB

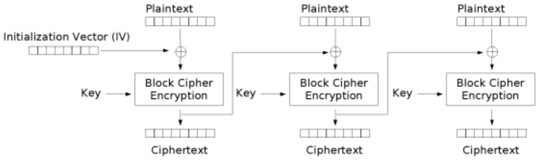

chained modes

ECB FAIL



- Still looks like Tux to me
- Block-level patterns (or bigger) still visible in encrypted output

What Is CBC Mode?



Cipher Block Chaining (CBC) mode encryption

- Most common chained block cipher mode
- The output of the block cipher function is XORed with the next plaintext block
- First plaintext block is XORed with an *Initialization Vector (IV)*
- This makes each ciphertext unique

CBC Mode Fixed IV Flaw

- Typically sites use same key for every user
- You make mistake of using fixed IV for every entry
- This means two of the three inputs are identical, so:
- Identical plaintexts encrypt to identical ciphertexts
- What if you were encrypting a password database?

Encrypting When You Need Integrity Protection

- Most people who think of crypto think of encryption.
- Your session IDs probably *don't* need to be confidential
- Your session IDs probably *do* need to be returned unmodified
- Your session IDs probably *do* need to be unforgeable
- Encryption is almost *always* wrong for this (see my other presentation)

Implications of No Integrity Protection

- Fiddling with ciphertext usually corrupts at least one block
- If you're *lucky*, a randomly-corrupted block will yield a syntactically-invalid plaintext string

Quote

"Shallow men believe in luck. Strong men believe in cause and effect."
– Ralph Waldo Emerson

No Integrity Protection Fail

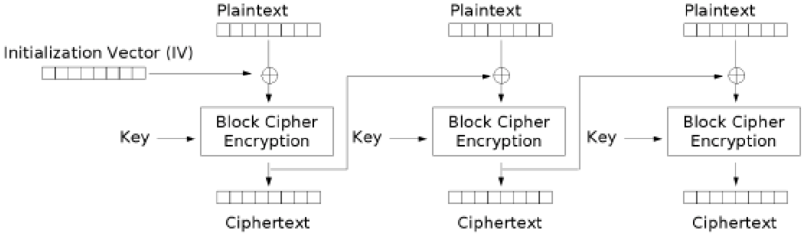
Message Authentication Codes

- Want a way to verify data haven't been tampered with
- Hash isn't enough; could tamper with data and recompute hash
- We need something like a "keyed hash"
- Several attempts made before finding a secure solution

CBC-MAC

- Encrypt the message in CBC or CFB mode
- Hash is last encrypted block, encrypted once more for good measure
- CBC form specified in ANSI X9.9, ANSI X9.19, ISO-8731-1, ISO 9797, etc.
- Let's review CBC mode

CBC Mode



Cipher Block Chaining (CBC) mode encryption

- Can anyone guess the problem in using last ciphertext for MAC?

CBC-MAC Vulnerability

- Recipient must know the key
- Recipient can decrypt the MAC with the key
- Block ciphers are *reversible*
- Therefore, can create *preimages* with the same MAC value
- Not really a big deal if you're the sender and recipient

CBC-MAC Fail

- Reversible - no *preimage resistance*

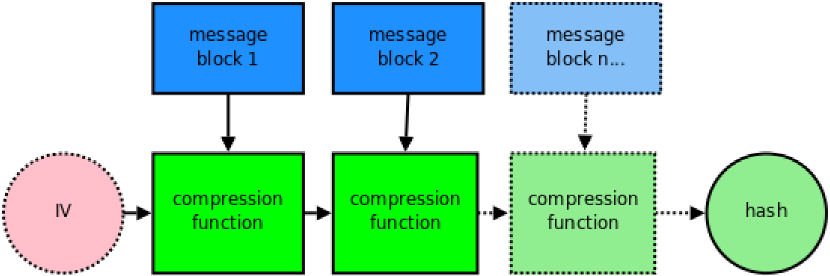
Bidirectional MAC

- First compute CBC-MAC of message
- Then compute CBC-MAC of blocks in reverse order
- Broken by C.J. Mitchell in 1990
- Exact vulnerability is unclear, but appears to suffer from same problem as CBC-MAC

One-Way Hash Function MAC

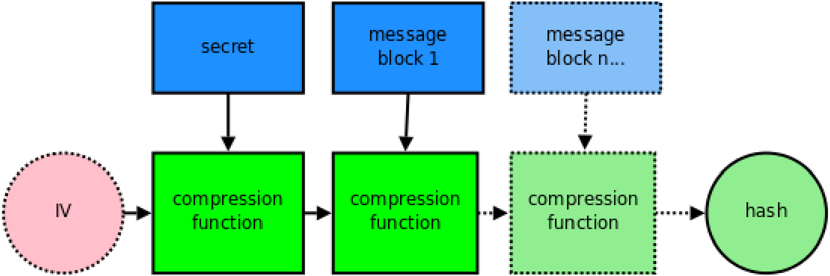
- Alice and Bob share key K
- Alice wants to send Bob a MAC for message M
- $MAC = H(K + M)$
- What is wrong with this method?

Iterative Hash Function Construction



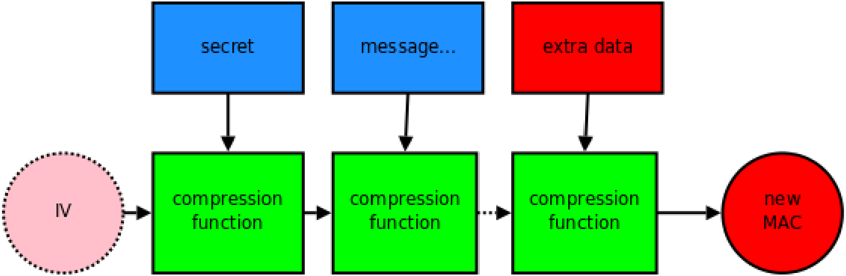
Compression function is one-way, IV is usually fixed

One-Way Hash Function MAC



Assume secret is one block, message is one or more blocks; where is the flaw?

One-Way Hash Function MAC Broken

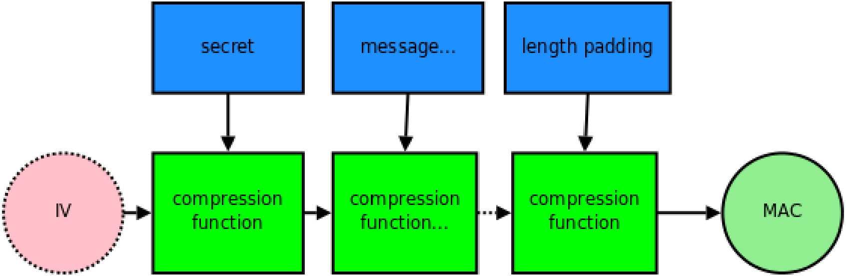


Flaw

Anyone can tack data onto the end of the message and generate a new MAC

One-Way Hash Function MAC

With Merkle-Damgaard Strengthening

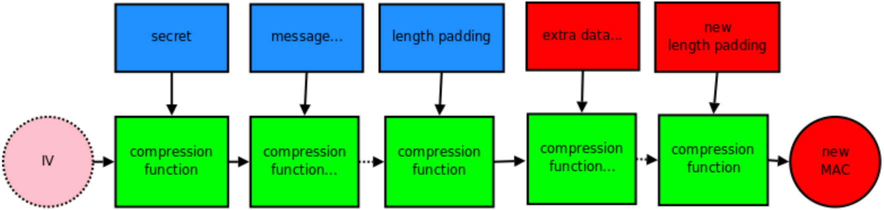


Hashes can be strengthened against length-extension attacks by encoding the length as padding

See any problems with this?

One-Way Hash Function MAC Broken

With Merkle-Damgaard Strengthening



Flaw

Anyone can still tack data and a new length onto the end of the message and generate a new MAC

Netifera found this vuln in Flickr API in Sep 2009

Questionable One-Way Hash Function MACs

- Prepend message length - cryptographer B. Preneel is suspicious
- Better to put secret key at end of hash: H(M + K) - this has B. Preneel suspicious too
  - Collisions in hash make this MAC malleable
- Still better is $H(K+M+K)$ or $H(K_1+M+K_2)$ - Preneel still finds suspicious

One-Way Hash Function MAC Fail

- Many have tried
- Few win

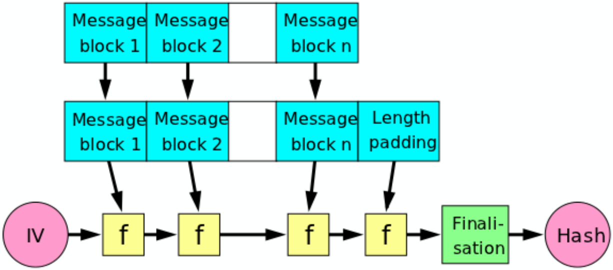
Other One-Way Hash Function MACs

- $H(K_1 + H(K_2 + M))$
- $H(K + H(K + M))$
- $H(K + p + M + K)$ where p pads K to full message block
- Concatenate 64 bits of key with *each* message block in hash

All of these *seem* secure but there's no proof
Given the history it's wise to be skeptical

Aside: Stronger Hashes

Full Merkle-Damgaard Construction



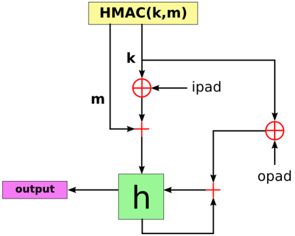
If finalization function is one-way, length extension attacks against the hash are not possible.

HMAC

$$HMAC_K = h((K \oplus opad) + h((K \oplus ipad) + m))$$

opad = 0x5c5c5c...5c5c

ipad = 0x363636...3636



Doesn't make sense but comes with a proof of correctness.

HMAC Win

Deriving Multiple Keys From One

Standard way is to seed a PRNG, but they are the least well-analyzed crypto primitives.
Here is a way to use HMAC to do it.

making two keys from one

Given secret s, derive two keys ($k^1$and k^2) from it
$k^1 = HMAC(s, "1")$
$k^2 = HMAC(s, "2")$
...
Given either or both keys will not help you retrieve s or any other k derived from s

No Public-Key Needed

	symmetric	asymmetric
encryption	cipher	PK encryption
integrity	MAC	dig sig

- All parameters sent to web browsers come back to your server, so you don't need asymmetric crypto
- Except HTTPS/SSL/TLS of course, but that is all cookbook

Wordpress Cookie Integrity Protection Setup

Wordpress Cookie Construction

- let | be a seperator character of some kind
- authenticator = USERNAME + | + EXPIRY_TIME + | + MAC
- MAC = HMAC-MD5$_K$(USERNAME + EXPIRY_TIME)

USERNAME The username for the authenticated user

EXPIRY_TIME When cookie should expire, in seconds since epoch

Any guesses as to the flaw?

Wordpress Cookie Integrity Protection Vulnerability

The Flaw

- $\text{HMAC-MD5}_K(\text{USERNAME} + \text{EXPIRY\_TIME})$

- HMAC didn't put a delimiter between username and expirytime

Wordpress Cookie Integrity Protection Attack

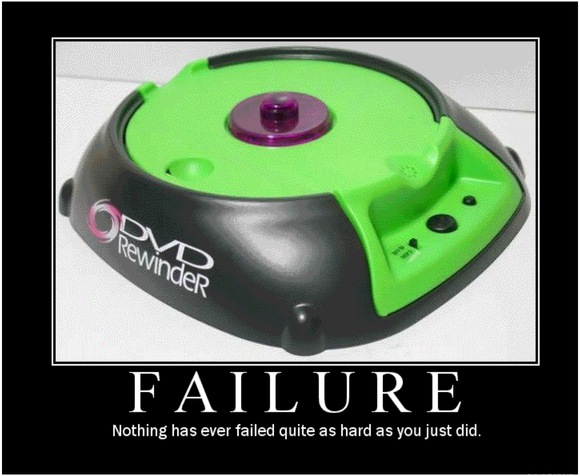
- Ask site to create authenticator for username "admin0", then create forged authenticator:

Forged Authenticator

- authenticator = "admin" + | + EXPIRY_TIME_1 + | + *HMAC-MD5*$_K$*("admin0"+*EXPIRY_TIME_2*)*
- "admin" + EXPIRY_TIME_1 = "admin0" + EXPIRY_TIME_2

- The HMAC-MD5 block was from the admin0 account cookie
- EXPIRY_TIME_1 is the same as EXPIRY_TIME_2 but lacks a leading zero
- Due to second equality, MAC verifies properly
- Tricky attack that is solved by using unambiguous formatting

Wordpress Fails It!



- Crypto payloads need unambiguous representations
- That's why we have ASN.1, but it would be overkill

Don't Do This

- *Don't* use ECB mode
- *Don*'t use stream ciphers such as RC4
- *Don't* use MD5 hashes, or even SHA-1
- *Don't* reuse keys for different purposes
- *Don't* use fixed salts or IVs
- *Don't* roll your own cipher
- *Don't* rely on secrecy of a system
- *Don't* use guessable values as random numbers or PRNG seeds

Suggestions

- Keep it simple as it can be but no simpler
- Understand the cryptographic properties of the tools
- Assume adversary knows all but the keys
- Always strive for unambiguity in your plaintexts and ciphertext blocks

Specific Suggestions

- When in doubt, use:
  - AES256 mode for encryption (CBC mode unless you're mixing data sources)
  - HMAC-SHA512 for integrity protection
  - SHA-512 with salt for hashing
  - PBKDF2 for stored passwords or key derivation
  - /dev/urandom on Unix
  - RtlGenRandom/CryptGenRandom from ADVAPI32.DLL on MSWin

For Further Reading I

- The Cookie Eaters
  http://cookies.lcs.mit.edu/
- OWASP5037 - Cryptography for Penetration Testers by Chris Eng
  http://video.google.com/videoplay?docid=-5187022592682372937&hl=en
- Cryptography - Theory and Practice by Steve Weis
  https://www.youtube.com/watch?v=IzVCrSrZIX8
- Crypto Strikes Back! by Nate Lawson
  https://www.youtube.com/watch?v=ySQl0NhW1J0