Nate Lawson

June 9, 2009

Yahoo! Security Week

My background

- Root Labs founder
  - Design and analyze security products
  - Focused on:
    - Embedded and kernel security
    - Software protection, DRM
    - Crypto and protocols
- Cryptography Research
  - Reviewed numerous products for flaws
  - Co-designed Blu-ray disc content protection layer
- Infogard Labs
  - Review crypto products for the FIPS 140 program
- IBM/ISS
  - Original developer of RealSecure IDS

We all use AES-CBC

- Crypto is deceptively simple and attractive
  - Block ciphers, public key, etc. make sense conceptually
  - Books like Applied Cryptography make it accessible
  - Proofs of security, brute-force strength claims are appealing
- Of course, no one implements their own ciphers
  - Most got the message that custom ciphers are bad
  - Many low-level crypto libraries to choose from
    - Java crypto API, BouncyCastle
    - Microsoft CryptoAPI, .NET Cryptography
    - OpenSSL, Crypto++
- Only remaining decision is 256 vs. 128 bit keys?

If you're typing "A.E.S.", you're doing it wrong

- Crypto is extremely fragile
  - Review is 2x the development cost
    - If you design it, you don't get to review it also
  - Can't be secured by testing
  - When it fails, you get to change out root private keys

CRYPTO FAIL!

root labs

How to avoid implementing crypto

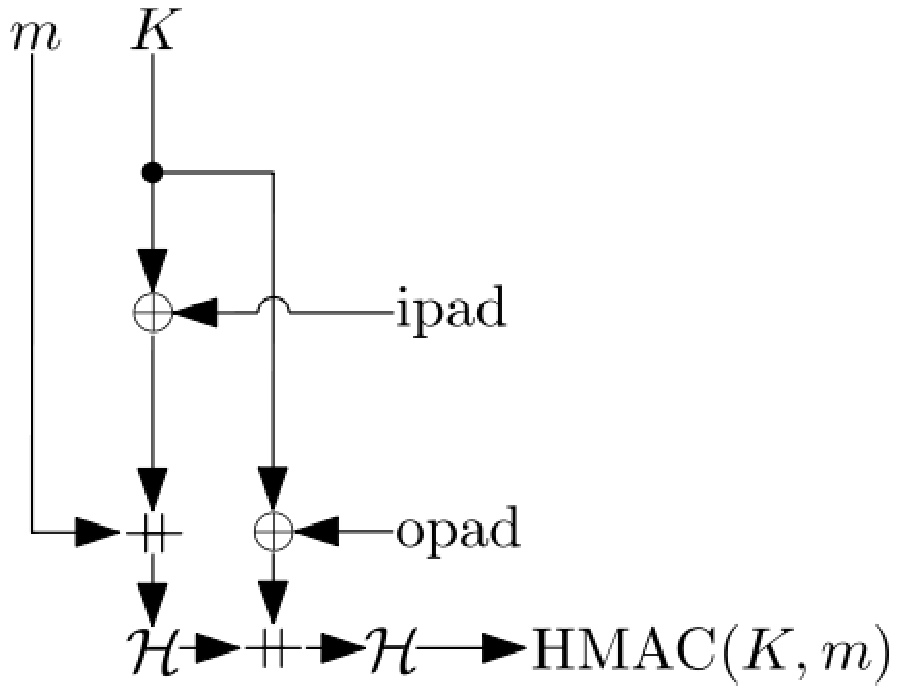
- What to do instead
  - Use SSL for transport
  - Use GPG for protecting data at rest (GPGME lib)
- If your design doesn't seem to fit that model, rework it until it does
- If you can't, use a high-level library
  - GPGME (LGPL)
  - Keyczar (Apache)
  - Gutmann cryptolib (commercial)
- Don't use low-level libraries directly
  - OpenSSL, Crypto++
  - Java crypto, BouncyCastle
  - .NET System.Security.Cryptography

Warning signs: bad crypto ahead

- Obvious confusion
  - Using a MAC to "sign" data
  - "Decrypting" a signature to verify it
  - Multiple names/terms for the same field/operation
  - Overly focused on salts as a key security feature
  - The words "rainbow tables"
- Arguments why non-standard usage is ok, based on…
  - Key size
  - Lack of attacker knowledge
  - A blog post

Warning signs: bad crypto ahead

- Encrypting multiple blocks with RSA
- Insistence that the IV must be kept secret
- Spec diagrams from Wikipedia



http://en.wikipedia.org/wiki/HMAC

root labs

How to fix Javascript crypto

- Don’t do it; it makes no sense
  - Where did the browser get the code from?
    - The webserver
  - Over?
    - SSL (we hope)
  - And the keys that the Javascript uses?
    - Um, the same webserver

- You have SSL on the client and server, use it
  - Anything you roll yourself is going to be worse
  - Remove the words “Javascript crypto” from your vocabulary (ahem, Meebo)

“Maybe they want to sign something, without having to upload it to the server, say. Or maybe they just don't want to burden the server with tons of crypto. There's lots of good reasons to do crypto without reinventing SSL.“

-- Dave Aitel on Javascript crypto

Upgrading encryption on-the-fly

- Fixing an old, broken cookie scheme
  - Old: CBC-encrypted cookie but no integrity protection
  - New: + HMAC protection of encrypted cookie

- Bright idea: “We’ll upgrade user cookies!”
  - Decrypt the data, check length of result
  - Detect old cookie based on version field
  - Set new version, encrypt with new key, add HMAC
  - Return it to the user

Bad PRNG or no seeding

- Default "random" PRNGs not a good choice
  - Cryptographic generator has different requirements
    - Constantly re-seeded with new entropy
    - Impossible to guess or even perturb internal state
  - Don't use:
    - Python random (use random.SystemRandom)
    - java.util.Random (use java.security.SecureRandom)
    - .NET System.Random (use System.Security.Cryptography. RandomNumberGenerator)
- Cold boot case
  - Server just rebooted and you're already getting requests
    - Is there enough entropy?
    - Are you replaying IVs, challenge values, etc.?
  - Wait, who in your company is responsible for this?

DSA fails if random value known

- If the random challenge value k is known, it reveals your entire private key
- DSA signature (r, s) is:

  $r = g^k \bmod p \bmod q$
  $s = k^{-1} (H(m) + x*r) \bmod q$

- If you know k, you can calculate x as follows:

  $x = ((s * k) - H(m)) * r^{-1} \bmod q$

- Remember that Debian PRNG bug?
  - Every DSA key used on a vulnerable system for a single signature is forever compromised
- Surprise!
  - Knowledge of only a few bits of k and multiple signatures is sufficient
  - Hidden number problem (Boneh, others)

Block cipher modes of operation

- If you know ECB from CBC, give yourself a hand
- Now the following
  - OFB, CFB, CTR
    - Stream cipher constructions
    - Bitwise malleability
    - Fatal if counter or output reused
  - CCM, EAX, GCM, OCB
    - Authenticated encryption
    - Provide integrity protection + encryption
    - Fatal if IV reused
- Decipher this one
  - “TDEA Electronic Code Book (TECB) Mode”
  - Seen in: docs for USB flash key

Counter duplication in CTR mode

- You painstakingly made sure to not duplicate the counter on the client
  - Using a counter and saving its state in a variable
  - You even persisted it to disk in case of a reboot
    - Or force a renegotiation if you have that option
- What about the server?
  - If using a shared key, you have keystream reuse if the server’s counter overlaps the client's

Integrity protection with only a hash

- Goal: prevent modification of a cookie
- Solution
  - Send SHA1(key || data)
  - User can't create the checksum because they don't know the secret, right?
- Length extension attack
  - See SHA1 init and final operations below:
- Use HMAC instead
  - Other ad-hoc constructions also vulnerable

Initialize variables:
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0
…
Produce the final hash value:
return (h0 || h1 || h2 || h3 || h4)

Padding error oracle with CBC encryption

- Plaintext is HMACed, padded to block size, CBC-encrypted
  - Example for AES, message length 14 bytes
    - Block = p[0], … p[13], 0x01, 0x01
- Server returns two different errors
  - Padding_Incorrect: pad byte was wrong value for specified total length
  - Integrity_Failure: padding valid but message modified
- Now you can decrypt bytes
  - Set a block with different guesses for last byte
  - If correct guess, you'll get Integrity_Failure error
  - Repeat for other bytes
- Seen in: OpenSSL (Bodo Moeller)

Lack of structure in signed data

- You have to know exactly what you're signing!
  - Think of signed data as a command, typed into a global root shell, on every cluster in your company
- Amazon Web Services v1
  - Allowed the client to sign a URL
    1. Split the query string using '&' and '=' delimeters
    2. Concatenate all the key/value pairs into a single string (key1 || value1 || key2 || value2)
    3. Protect it with HMAC-SHA1(key, data)
- No structure in data means the following are equivalent
  - …?GoodKey1=GoodValue1BadKey2BadValue2
  - …?GoodKey1=GoodValue1&BadKey2=BadValue2
- Seen in: many SSO (single sign-on) designs

Padding is not something to ignore

- PKCS#1 pads messages to be signed as follows
  - SigData = 00 01 FF … FF 00 <SHA1-OID> <SHA1-HASH>
  - Then, signature is RSA-Exp(PrivKey, SigData)
- Signature verification seems simple
  - Process the sig with RSA-Exp(PubKey, SigData)
  - Find the hash and compare to the hash of the message
- Now attacker can create forgeries
  - For 2048 bit key and 160-bit SHA hash, there are 2^{1888} inputs that will match
  - With small public key, straightforward algebra to calculate a colliding signature
- Every bit of padding data must be strictly checked
- Seen in: OpenSSL (+Opera, Konqueror), Firefox, Nintendo Wii

Bare (no hash) signatures

- Just use RSA directly, why hash first?
  - Remember warning about lack of structure?
- Attack
  - Choose some small primes
  - Multiply them together to form your message
  - Get it signed
  - Now forge messages composed of any combination of those primes
- Fun fact: same property that allows blinding
- Seen in: embedded microcontroller

“Encrypting” with RSA private key

- Example
  - Device has an RSA public key to verify signature on firmware updates
  - Idea: keep the public key secret and use it to decrypt the updates also
- RSA is incompatible with block cipher thinking
  - You can’t “encrypt” with a private key and keep the public key secret
  - An attacker can always figure out the public key (e, n) from observing two signatures

```
for e in [3, 5, 7, … 65537]:
    n ~= GCD(sig_1^e – m_1, sig_2^e – m_2)
    if m_1^e mod n == sig_1:
        break
```

Comparing hash string mistakes

- Verifying an RSA signature
  - RSA-Exp(PubKey, SigData)
  - Verify all the padding data as you should
  - return (0 == strncmp(userHash, myHash, 20));
- Attack
  - Create 256 messages, each slightly different
  - Once SHA-1(message) == {00, …}, you win!

- Seen in: Wii (tmbinc)

SRP parameter validation

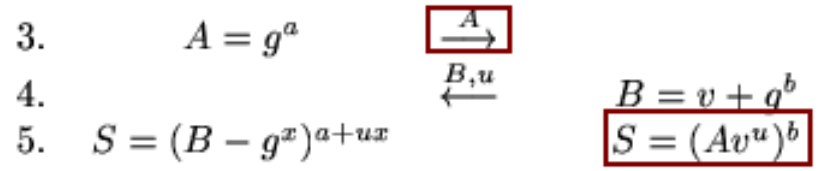
- SRP gives secure password-based key agreement
- Seems easy, right?

	Carol		Steve
1.		$\xrightarrow{C}$	(lookup s, v)
2.	$x = H(s, P)$	$\xleftarrow{s}$	
3.	$A = g^a$	$\xrightarrow{A}$	
4.		$\xleftarrow{B,u}$	$B = v + g^b$
5.	$S = (B - g^x)^{a+ux}$		$S = (Av^u)^b$
6.	$K = H(S)$		$K = H(S)$
7.	$M_1 = H(A, B, K)$	$\xrightarrow{M_1}$	(verify M_1)
8.	(verify M_2)	$\xleftarrow{M_2}$	$M_2 = H(A, M_1, K)$

- Like all public-key crypto, very sensitive to choice of values

SRP parameter validation bug

- What if client sends A=0?



- Hmm, server does $(A * foo^{bar})^{baz} = 0$
  - And there are other bad parameters with similar results

- Seen in: samhain host integrity tool (Ptacek)

I probably ran out of time here

- DH small subgroup confinement attack (Tor, IKE)
  - Sending magic values lets you choose the server's key (similar to SRP attack)
- Encrypting a CRC for an integrity check (SSH v1)
- Fault injection during RSA signing (Pay TV smart card glitching)
- Timing and other side channel attacks
  - Guessing correct HMAC by time-to-reject (Keyczar, Xbox360)
  - Instruction/data and branch predictor cache leaks (everyone)

Conclusion

- Crypto is brittle (easy to mess up)
  - And when it fails, the results are spectacular
- So don't build your own crypto
  - Use SSL for transport
  - Use GPG for protecting data at rest (GPGME lib)
- If your design doesn't seem to fit that model, rework it until it does
- If you can't, use a high-level library
  - Keyczar or cryptolib
- Always get a third-party review of any crypto code
  - If you had to design your own (i.e., directly using a low-level library), much more extensive review required

Contact info

For more detailed articles on these topics, please see my blog linked at: http://rootlabs.com

Questions?

Nate Lawson
nate@rootlabs.com