Embedded Device Cryptography in the Field

Alex Kropivny

January 5, 2015

wurldtech
A GE Company

wurldtech
A GE Company

Introduction

Motivation

Who am I?

Senior security analyst at a device assessment team.

- [ ] cryptographer
- [x] reverse engineer
- [ ] hat owner

Want to one day become a full stack developer. Still not done counting all the layers.

Device Assessment Team?

Security assessments of *embedded devices*[1] and *software systems*[2] that use them.

- Design reviews and source code audits for manufacturers.
- Black box reverse engineering for major end users.

Automation, smart grid, medical industries - disclosure left up to clients.[3]

---

[1] Catch-all term for magic black boxes that do stuff.
[2] Heterogeneous networks that make security *fun*.
[3] Any vulnerabilities shown in these slides aren't theirs.

Talk Scope

For simplicity, let "embedded devices" be:

- 1 kB - 1 MB program memory.
- 1 MHz - 100 MHz clock frequency.
- No money spent on tamper resistance or DRM.
- No Linux/Windows/...
- No OpenSSL/GnuPG/Bouncy Castle/...

Not all bad news:

- Small attack surface!
- Single purpose!
- Analysis is easy!

What Qualifies as a Break?

Our team has to be pragmatic. If it's not exploitable against a real-world system, it's not a result.

Attack	Valid
Remote code execution	Always
Control or reconfiguration	Often
Denial of service	Rarely
Privacy	Very Rarely

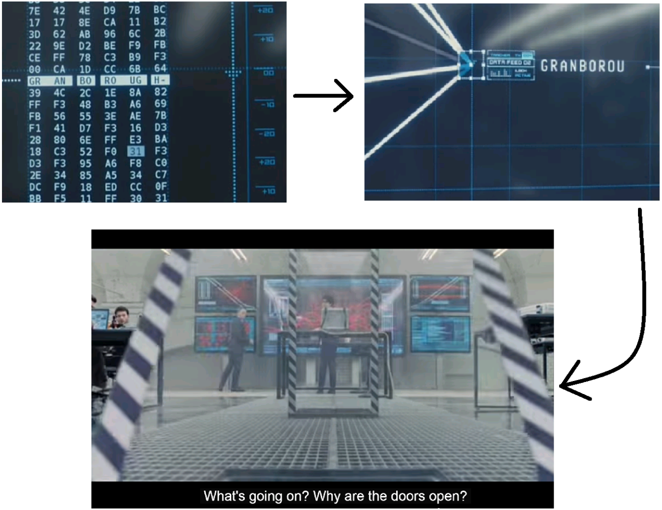
Pop Quiz

Q: What fraction of cryptographic constructions do we find valid “results” in?

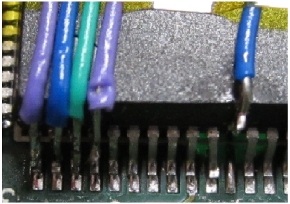
wurldtech
A GE Company

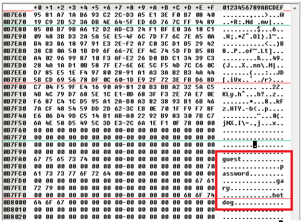
State of Affairs

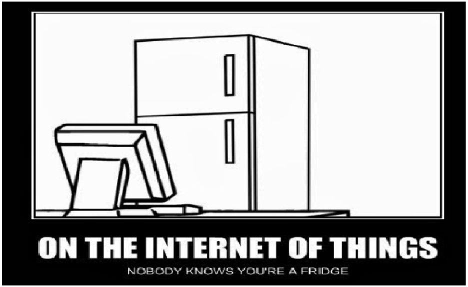
Hollywood SCADA Hacking



Actual SCADA Hacking







~~Vulnerabilities~~ Surprise Features

System with no threat model can't be insecure, only surprising.

Embedded Device Cryptanalysis


```
int *__cdecl HTCA_GenSymmetricKeyA(int *size, int a2, int *error)
{
  int *key; // esi@1
  int *result; // eax@1
  DWORD seed; // eax@3
  unsigned int i; // edi@3

  *error = 0;
  key = (int *)j_malloc(32);
  result = 0;
  if ( key )
  {
    *key = 0;
    key[1] = 0;
    key[2] = 0;
    key[3] = 0;
    key[4] = 0;
    key[5] = 0;
    key[6] = 0;
    key[7] = 0;
    seed = GetTickCount();
    j_srand(seed);
    i = 0;
    if ( *size )
    {
      do
        *((_BYTE *)key + i++) = j_rand();
      while ( i < *size );
    }
    result = key;
  }
  else
  {
    *error = 5;
  }
  return result;
}
```

Embedded Device Cryptanalysis


```
public static string DecryptString(string strText)
{
   return DecryptString(strText, "ras258");
}


public static string DecryptString(string strText, string key)
{
   string str = string.Empty;
   try
   {
      byte[] buffer = new MD5CryptoServiceProvider().ComputeHash(Encoding.ASCII.GetBytes(key));
      TripleDESCryptoServiceProvider provider = new TripleDESCryptoServiceProvider {
         Key = buffer,
         Mode = CipherMode.ECB
      };
      byte[] inputBuffer = Convert.FromBase64String(strText);
      str = Encoding.ASCII.GetString(provider.CreateDecryptor().TransformFinalBlock(inputBuffer, 0, inputBuffer.Length));
   }
   catch (Exception)
   {
   }
   return str;
}
```

Coping Mechanisms

Approach

If it's stupid and it works, it's not stupid.

Blame is the enemy of safety. Focus should be on understanding how the system behavior as a whole contributed to the loss and not on who or what to blame for it.[4]

[4] Engineering a Safer World: Systems Thinking Applied to Safety

Talk Outline

- What are major uses and threat models we've seen?
- How do their implementations fail? (Vulnerabilities rated from $\star$ to $\star\star\star\star\star$ based on frequency seen.)
- If possible, why does the failure occur?

wurldtech
A GE Company

Indefensible

wurldtech
A GE Company

Local Attacks

Local Attacks

Against *low-cost* devices not hardened against them, attacks range from easy to doable:

- Side channels
- Fault injection
- Decapping and probing + fault injection

Deleting keys on tamper would be nice, but:

- One-way operations that brick the device are scary to deploy.
- Requires an internal power supply, which adds *cost*.
- Tamper detection for one device is easy; for two or more, extremely hard.

Local Attacks

If a device is widely available to attackers, hardware compromise in the large can be assumed.

- On widely deployed devices, shared secrets are massive central points of failure.
- In an ideal world, compromise via local access does not give attacker any more capabilities than they already have.
- Good bang-for-buck measures exist to make local attacks harder do exist. (Disabling read access to internal memory, burning fuses.)

User As Threat (DRM)

DRM/smart card technologies make well-funded attempts to defend against some local attacks.

- Higher cost per chip!
- Cost of comparing security of different vendors/models high.

Better off spending resourses on system architecture that avoids shared secrets and distrusting the user, if possible.

Trust Relationships

wurldtech
A GE Company

Trust Relationships

The following will still be trusted:

- Manufacturer signing keys.
- Development infrastructure.
- Hardware and initial firmware bringup supply chain.

Often, use of cryptography merely shuffles trust around the system, but does not eradicate it.

Use Cases

Factory Testing

Production Line Testing

Needed to ensure all device functionality works as intended.

Most generic way to do it is via arbitrary read/write primitives to memory and registers.

Factory Commands

Manufacturers want access to peek/poke/jump primitives on the device for:
1. Factory testing. 2. Failure analysis.

Production Line Testing

Vuln (⋆⋆⋆⋆⋆)

The manufacturer authentication secret can be recovered from the target device. Kerkhoff's Principle is not common knowledge.

Vuln (⋆⋆⋆⋆⋆)

The manufacturer authentication secret does not use cryptographic primitives.

Since the functionality is only used in what we assume are completely trusted environments, the most trivial logon mechanism would suffice.

- $K_{unique} = MAC_{K_{master}}(serialnum)$
- Burn K_{unique} at fab.
- Compromise of one K_{unique} does not affect other devices.

The "trusted environment" assumption may be worth testing. . .

Production Line Testing

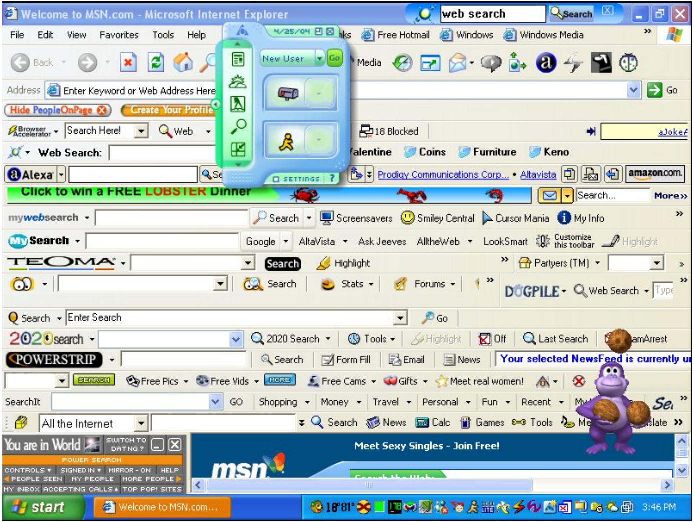


Figure 1: Typical Windows XP machine, courtesy of http://www.windows-noob.com/review/ie7/

Firmware Upgrades

Firmware Upgrade Security

Your device has firmware upgrades available for download.

Oh no! People can clone your device!

```
// chosen by fair dice roll.
// guaranteed to be random.
static const uint8 aesKey[KEY_BLENGTH] = {
  // This dummy key must be replaced by a randomly generated key that is kept secret.
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D,
};
```

Vuln (⋆⋆⋆⋆⋆)

Symmetric encryption key shared across many devices.

Firmware Upgrade Security


```
procedure TfrmAutoCode.FormCreate(Sender: TOb
begin
  edtEncryptKey.Text := 'D0 94 3F 8C 29 76 15
  edtDecryptKey.Text := 'D0 94 3F 8C 29 76 15
end;

procedure TfrmAutoCode.btnRandomKeyClick(Send
var
  i: Integer;
  s: string;
begin
  Randomize;
  s := '';
  for i := 0 to 31 do
  begin
    s := s + IntToHex(Random(256), 2) + ' ';
  end;
  edtEncryptKey.Text := s;
  edtDecryptKey.Text := s;
end;
```

```
if b then
  Writeln(f, '//¶¨ÒåÃÜÔ¿: 256Î»')
else
  Writeln(f, '//Define decrypt key: 256b');
Writeln(f, 'unsigned char DecryptKey[32] ={');
for i := 0 to 31 do
  buf[i] := 0;
s := edtDecryptKey.Text;
if Length(s) > 32 * 3 then
  SetLength(s, 32 * 3);
for i := 0 to (Length(s) + 2) div 3 do
  buf[i] := CharToByte(s[i * 3 + 1], s[i * 3 + 2]);
for i := 0 to 31 do
begin
  if (i mod 8) = 0 then
    Write(f, ' ');
  Write(f, ' 0x', IntToHex(buf[i], 2));
  if i <> 31 then
    Write(f, ',');
  if (i mod 8) = 7 then
    Writeln(f);
end;
Writeln(f, '};');
```

// chosen by fair dice roll.
// guaranteed to be random.

Figure 2: Firmware secured!

We can now reference “strong military-grade encryption” in our marketing materials.

Firmware Upgrade Security

Vuln (⋆⋆⋆⋆)

Firmware upgrade is encrypted with a symmetric key, but not authenticated in any way.

Vuln (⋆⋆)

Constant initialization vector.

Vuln (⋆)

ECB mode.

What does that actually mean?

Firmware Upgrade ~~Security~~ Confidentiality

Firmware Confidentiality

Manufacturer doesn't want firmware upgrade files to leak firmware contents.

- Key sharing is okay - by the time the key is extracted, so is the data it's protecting.
- IV reuse is more tricky, and depends on block cipher mode. Getting useful plaintext from one or two images under a stream cipher mode is tricky.
- Lack of authentication enables active attacks leading to firmware extraction:
  - ECB block swaps to nuke memory lockout flags.
  - Malleability to morph known code regions into dumper stubs.

Firmware Upgrade ~~Security~~ Authenticity

Firmware Authenticity

Manufacturer wants code execution on the device for factory testing and failure analysis.

Symmetric key re-use becomes critical.

Some local-only bypass vulns pop up.

Vuln ($\star$)

Time of check time of use between authentication and decryption passes. (Requires local access to external memory.)

Vuln ($\star$)

Expanded key remanence. (Requires local access to RAM, even if flash is inaccessible.)

Firmware Upgrade ~~Security~~ Authenticity

Mitigation comes down to two options:

1. Unique symmetric authentication key per device - significant extra key management infrastructure and network bandwidth needed.
2. Asymmetric signature - significant expertise needed for implementation.

When combined with the typical firmware upgrade challenge of not bricking the device, either one is non-trivial.

Wireless Protocols

Bolt On Crypto

Wireless Channel Encryption

Manufacturer wants to encrypt a communication channel without bloating an existing frame format with extra nonce/MAC data.

There's a variety of AEAD modes to choose from, but...

Vuln (⋆⋆⋆)

No message authentication.

Vuln (⋆⋆)

CRC-then-encrypt under a streaming block cipher mode confused for message authentication.

Vuln (⋆⋆)

Fixed IVs or ECB.

Bolt On Crypto (Modern AEAD Support Edition)

Vuln (⋆⋆⋆)

Least secure methods are the default/easiest to implement methods.

The **AES encryption/decryption core** allows the user to encrypt and decrypt data using the AES algorithm with 128-bit keys. The AES core also supports ECB, CBC, CFB, OFB, CTR, and CBC-MAC, as well as hardware support for CCM.

```
// AES Modes
#define CBC           0x00
#define CFB           0x10
#define OFB           0x20
#define CTR           0x30
#define ECB           0x40
#define CBC_MAC       0x50
```

+

```
void ssp_HW_KeyInit( uint8 *AesKey )
{
  AES_SETMODE(ECB);
  AesLoadKey( AesKey );
}
```

Figure 3: "High level" C API defaulting to ECB mode on CCM-supporting hardware

802.15.4 support is driving accessible CCM implementations.

Device Pairing

Wireless Device Pairing

Users must be able to pair new devices to their phone with minimal interaction.

Vuln (⋆⋆⋆)

Pairing broken by passive attacker due to using same channel.

Vuln (⋆)

Pairing broken by active MITM attacker due to lack of authentication.

Out of band channels don't get used much, strangely.

- Users can't be trusted to type in keys.
- QR codes inspire hate.

Device Pairing

Ideally, method would allow security-conscious users to put in extra effort while working instantly in low-risk cases.

```
/**
 * Start the bonding (pairing) process with the remote device using the
 * Out Of Band mechanism.
 *
 * <p>This is an asynchronous call, it will return immediately. Register
 * for {@link #ACTION_BOND_STATE_CHANGED} intents to be notified when
 * the bonding process completes, and its result.
 *
 * <p>Android system services will handle the necessary user interactions
 * to confirm and complete the bonding process.
 *
 * <p>Requires {@link android.Manifest.permission#BLUETOOTH_ADMIN}.
 *
 * @param hash - Simple Secure pairing hash
 * @param randomizer - The random key obtained using OOB
 * @return false on immediate error, true if bonding will begin
 *
 * @hide
 */
public boolean createBondOutOfBand(byte[] hash, byte[] randomizer) {
    //TODO(BT)
    /*
    try {
        return sService.createBondOutOfBand(this, hash, randomizer);
    } catch (RemoteException e) {Log.e(TAG, "", e);}*/
    return false;
}
```

Device Pairing

Vuln (⋆⋆⋆)

Least secure methods are the default/easiest to implement methods.

People have the assumption that key exchanges over short range communication won't be eavesdropped.

Other

Random Number Generation

Without `/dev/urandom`, there are three major failure modes.

Vuln (★★★★)

Default mode: forgot the CSPRNG, used an LCG.

Vuln (★★★)

Extreme deterministic mode: power cycle repeats output.

Vuln (★★★)

Extreme entropy mixing mode: low bits gathered from ADC or timer jitter used directly. "True Random Number Generator!"

Export Control Effects?

- System constructions show up that are too weird to have happened naturally.
- Sometimes a key size shows up that is too small to be anything but some relic of the past.
- (Manufacturer firmware upgrade keys are a better backdoor anyway.)

wurldtech
A GE Company

Conclusions

Conclusions

There is a large class of embedded devices that:

- Can run many common cryptographic primitives.
- Can use cryptography to secure common functionality in connected scenarios.

Current implementations tend to:

- Use standard primitives.
- Roll their own constructions.
- Re-invent the wheel over and over.

Failure Modes

Code uses NIST primitives and some NIST constructions, but gets copy-pasted together from vendor examples and stackoverflow.

1. Need libraries with userproof APIs and brand name recognition on more platforms.
2. Need embedded-friendly ECC libraries.
3. Need brand name recognition protocols for really boring embedded tasks.

Questions?