Experience with Verifying Cryptographic Software in C

Andrew D. Gordon, Microsoft Research and University of Edinburgh
Based on joint work with Mihhail Aizatulin (OU), François Dupressoir (OU), Jan Jůrjens (Dortmund) and David A. Naumann (Stevens)
Agenda

- Goal is to describe the state-of-the-art of verifying security properties of cryptographic software in C, and to suggest what’s next.

- Origins of crypto protocols; an example bug; verification of models; our work on verification of C code

- Next steps: download our tools; contribute to the Csec-Challenge; join our mailing list; come to CryptoForma
  - [http://groups.inf.ed.ac.uk/security](http://groups.inf.ed.ac.uk/security)
  - [http://research.microsoft.com/csec](http://research.microsoft.com/csec)
In “Using encryption for authentication in large networks of computers” (CACM 1978), Needham and Schroeder didn’t just initiate a field that led to widely deployed protocols like Kerberos, SSL, SSH, IPSec, etc.

They threw down a gauntlet.

“Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation. The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area.”

Principal A initiates a session with principal B
S is a trusted server returning public-key certificates eg \(\{A,KA\}\) for
NA,NB serve as nonces to prove freshness of messages 6 and 7
Example: Web Services Security

- "A web service is a web site intended for use by computer programs instead of human beings."
- Think XML not HTML

Service messages in SOAP format:
- Envelope/Header – addressing and security headers
- Envelope/Body – actual payload

The WS-* specs define basics like signatures and encryption, and protocols for functionality such as web-based single sign-on (as supported by Windows Cardspace) to counter phishing attacks

We studied these specs, found errors, and built security tools
A Rewriting Attack

UsernameToken assumes both parties know Alice’s secret password $p$

Without cracking the password, an adversary can alter the interpretation of this message by some early implementations

$hmacsha1(key, SignedInfo)$ where $key=psha1(p+nonce+created)$
Given this message, early versions of **Microsoft WSE** (Web Services Enhancements) would report a valid signature of the message **Body**.

But an XML message may contain two Body elements – we need to be explicit about which one is signed.

The indirect signature of the body, now hidden in **BogusHeader**, may still appear valid.
Crypto Code is Still Hard to Get Right

- Design & implementation errors lead to many serious security vulnerabilities: SAML, OpenSSL, ASP.NET
- Traditional crypto models miss most details
- Production code and design specs differ

We verify security on production code
ProVerif, the state of the art, around 2008

VERIFYING ABSTRACT MODELS OF PROTOCOLS
An Example Protocol

Authenticated RPC: RPC-enc

\[ A \rightarrow B: A, \{ \text{request}, k_{req} \}^{k_{AB}} \]
\[ B \rightarrow A: \{ \text{response} \}^{k_{req}} \]

Client: Received encrypted message:
6a64b21d6d93a65aead74fa820d7049fd661bd2a
9495deae59c528b51e4042cb10a47d507e42c1c
132a8855b5d8081c46197131

Client: Received and authenticated response:
Look out the window.

Server: Preparing response: Look out the window.

Server: Sending encrypted message:
6a64b21d6d93a65aead74fa820d7049fd661bd2a
9495deae59c528b51e4042cb10a47d507e42c1c
132a8855b5d8081c46197131
Authenticated RPC: RPC-enc

\[
\begin{align*}
A \rightarrow B & : A, \{\text{request}, k_{\text{req}}\}_{k_{AB}} \\
B \rightarrow A & : \{\text{response}\}_{k_{\text{req}}}
\end{align*}
\]

let A = event client_begin(request);
new kS1;
let var1 = conc1(clientID, E(kAB, conc1(request, kS1))) in
out(c, var1);
in(c, msg1);
in(c, var2);
event client_accept(request, D(KS1, var2)); 0.

let B =
in(c, msg2);
in(c, var12);
new response1;
event server_reply(fst(D(kAB, snd(var12))), response1);
let var13 = E(snd(D(kAB, snd(var12))), response1) in
out(c, var13); 0.

process ! new kAB; (!A | !B)
PhD work of Mihhail Aizatulin, paper at ACM CCS 2011

VERIFYING C CODE OF PROTOCOLS
int send_request(RPCstate * ctx)
{
    uint32_t m1_len, m1_e_len, full_len;
    unsigned char * m1, * p, * m1_e;
    m1_len = 1 + ctx->k_s_len + sizeof(ctx->request_len) + ctx->request_len;
    p = m1 = malloc(m1_len);
    memcpy(p, "p", 1);
    p += 1;
    * (uint32_t *) p = ctx->request_len;
    p += sizeof(ctx->request_len);
    memcpy(p, ctx->request, ctx->request_len);
    p += ctx->request_len;
    memcpy(p, ctx->k_s, ctx->k_s_len);
    full_len = 1 + sizeof(ctx->self_len) + ctx->self_len
    + encrypt_len(ctx->k_ab, ctx->k_ab_len, m1, m1_len);
    p = m1_e = malloc(full_len);
    memcpy(p, "p", 1);
    p += 1;
    * (uint32_t *) p = ctx->self_len;
    p += sizeof(ctx->self_len);
    memcpy(p, ctx->self, ctx->self_len);
    p += ctx->self_len;
    m1_e_len = encrypt(ctx->k_ab, ctx->k_ab_len, m1, m1_len, p);
    full_len = 1 + sizeof(ctx->self_len) + ctx->self_len + m1_e_len;
    send(&(ctx->bio), &full_len, sizeof(full_len));
    send(&(ctx->bio), m1_e, full_len);
}

stack m1_len ⇒ 1 + len(kS) + 4 + len(request)

stack p ⇒ ptr(heap6, 0)
stack m1 ⇒ ptr(heap6, 0)
heap 6 ⇒ 'p'
stack p ⇒ ptr(heap6, 1)
heap 6 ⇒ 'p'|len(request)
stack p ⇒ ptr(heap6, 5)
heap 6 ⇒ 'p'|len(request)|request
stack p ⇒ ptr(heap6, 5 + len(request))
heap 6 ⇒ 'p'|len(request)|request|kS
stack full_len ⇒ 5 + len(clientID)
+ encrypt_len(msg1)

stack p ⇒ heap7
stack m1_e ⇒ heap7
heap 7 ⇒ 'p'
stack p ⇒ ptr(heap7, 1)
heap 7 ⇒ 'p'|len(clientID)
stack p ⇒ ptr(heap7, 5)
heap 7 ⇒ 'p'|len(clientID)|clientID
stack p ⇒ ptr(heap7, 5 + len(clientID))
heap 7 ⇒ 'p'|len(clientID)|clientID|cipher1
stack m1_e_len ⇒ len(cipher1)

new fact: len(cipher1) ≤ encrypt_len(msg1)
cipher1 = E(key(clientID, serverID), msg1)
msg1 = 'p'|len(request)|request|kS
stack full_len ⇒ 5 + len(clientID)
+ len(cipher1)
generate IML:
out(c, 5 + len(cipher1) + len(cipher1));
generate IML:
out(c, 'p'|len(clientID)|clientID|cipher1);
<table>
<thead>
<tr>
<th>C LOC</th>
<th>IML LOC</th>
<th>outcome</th>
<th>result type</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple mac</td>
<td>~ 250</td>
<td>12</td>
<td>verified</td>
<td>symbolic</td>
</tr>
<tr>
<td>RPC</td>
<td>~ 600</td>
<td>35</td>
<td>verified</td>
<td>symbolic</td>
</tr>
<tr>
<td>NSL</td>
<td>~ 450</td>
<td>40</td>
<td>verified</td>
<td>computat.</td>
</tr>
<tr>
<td>CSur</td>
<td>~ 600</td>
<td>20</td>
<td>flaw: fig. 11</td>
<td>—</td>
</tr>
<tr>
<td>minexplib</td>
<td>~ 1000</td>
<td>51</td>
<td>flaw: fig. 12</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 10: Summary of analysed implementations.

```
read(conn_fd, temp, 128);
// BN_hex2bn expects zero-terminated string
temp[128] = 0;
BN_hex2bn(&cipher_2, temp);
// decrypt and parse cipher_2
// to obtain message fields
```

Figure 11: A flaw in the CSur example: input may be too short.

```
unsigned char session_key[256 / 8];
...
// Use the 4 first bytes as a pad
// to encrypt the reading
encrypted_reading =
    (unsigned int) *session_key) ^ *reading;
```

Figure 12: A flaw in the minexplib code: only one byte of the pad is used.
New: Computational Verification

- First security analysis of C code to target a verifier for the probabilistic computational model (ie, not assuming “perfect” crypto)
- Verify over 3000 LOC, more than any prior work on cryptographic code in C
- Our largest example, about 1000 LOC, independently written; during its analysis we uncovered a vulnerability now fixed by its author.
Model Extraction

- Allows automatic extraction of protocol model from code
  - Assumes protocol follows a single correct run, and any deviation should terminate immediately
  - Tools allows protocol designer to write $\pi$-calculus in C (!)
  - Verification shows the model is correct, but not the code, as it may follow other paths

- Future directions
  - Backpatch the code to terminate if it deviates from normal path
  - Scale to more examples eg SmartMeter, openSSL handshake
Towards Full Verification

• Proves memory safety and symbolic security of C code
  – PhD work of Francois Dupressoir, paper at CSF’11
  – Full verification based on the MSR VCC tool, but needs much more interactive effort than symbolic execution
  – Not preventing timing, power consumption, physical attacks

• Major case-study
  – Working this spring with Trusted Computing Group on next version of TPM chip – using stylized ANSI-C as a normative “Machine+Human-Readable Specification”
The Csec Challenge

• Surprisingly few bite-sized examples ie few hundred LOC
  – Several multiple KLOC crypto codebases eg openSSL, PolarSSL
• The absence of small examples hinders research community
• In 2011, we launched the **Csec Challenge**, a collection of challenge problems, including source code, intended security properties, and the results obtained by verification tools.
  • At launch we have two collections, and we aim to grow
    – **Csec-protocols**: Csur, NSL, RPC, RPC-enc
    – **CryptoKiX**: implements a PKCS#11 software token
    – Contributions from OU, ENS Cachan, Venice, CMU
• See [http://research.microsoft.com/csec-challenge](http://research.microsoft.com/csec-challenge)
Crypto Software in C

• The security of much critical infrastructure depends in part on crypto code in C, yet vulnerabilities continue to be found.

• Designers starting to appreciate reference implementations
  – eg, TCG “It has taken years to achieve good TPM interoperability”, need for “more precise language for describing TPM data structures and behaviors”

• What’s next? Verifying substantial crypto implementations remains a Grand Challenge, achievable in a few years

• Join our mailing list, come to CryptoForma, March 1-2
  – http://groups.inf.ed.ac.uk/security