Rule Based Static Analysis of Network Protocol Implementations

Octavian Udrea, Cristian Lumezanu, Jeff Foster
Univ. of Maryland, College Park
Motivation

• Network protocols must be reliable and secure

• Lots of work has been done on this topic
  – But mostly focuses on abstract protocols
  – ==> Implementation can introduce vulnerabilities

• **Goal:** Check that implementations match specifications
  – Ensure that the protocol we’ve modeled abstractly and thought hard about is actually what’s in the code
Summary of Results

• Ran on LSH, OpenSSH (SSH2 implementations) and RCP

• Found wide variety of known bugs and vulnerabilities
  – Well over 100 bugs, of many different kinds

• Roughly 5% false negatives, 38% false positives
  – As measured against bug databases
Pistachio Architecture

Existing documents and code

- RFC/IETF Standard
- C Source Code
- Bug Database

Rule-Based Specification ➔ Pistachio ➔ Evaluate Warnings ➔ Errors Detected

Pistachio ➔ Theorem Prover ➔ Errors Detected
A Toy Protocol

• *Alternating bit protocol*

1. Start by sending $n = 1$
2. If $n$ is received, send $n + 1$
3. Otherwise resend $n$
A Toy Protocol

int main(void) {
    int sock, val=1, recval;
    send(sock,&val,sizeof(int));
    while(1) {
        recv(sock,&recval,sizeof(int));
        if (recval == val)
            val += 2;
        send(sock,&val,sizeof(int));
    }
}

• **Alternating bit protocol**
  1. Start by sending $n = 1$
  2. If $n$ is received, send $n + 1$
  3. Otherwise resend $n$
A Rule Based Specification

Ø (empty hypothesis)

=>

send(_, out, _)
out[0..3] = 1

n := 1

• Alternating bit protocol

1. Start by sending $n = 1$
2. If $n$ is received, send $n + 1$
3. Otherwise resend $n$
A Rule Based Specification

recv(_, in, _)
in[0..3] = n
=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

• *Alternating bit protocol*

  1. Start by sending $n = 1$
  2. If $n$ is received, send $n + 1$
  3. Otherwise resend $n$
A Rule Based Specification

- recv( _, in, _ )
  in[0..3] ≠ n
  =>
  send( _ , out, _ )
  out[0..3] = n

- Alternating bit protocol
  1. Start by sending \( n = 1 \)
  2. If \( n \) is received, send \( n + 1 \)
  3. Otherwise resend \( n \)
Our Approach

• Use symbolic execution to simulate program execution
  – Track facts about program variables
  – Generated by assignments and branches

• Only simulate realizable paths
  – Test branch conditions using theorem prover

• Check rule conclusions hold
  – Using automatic theorem prover
1. Start by sending $n = 1$

Ø (empty hypothesis)

=>

send(_, out, _)

out[0..3] = 1

$n := 1$

Facts: {}

Matches the empty hypothesis
1. Start by sending $n = 1$

Ø (empty hypothesis) =>
send(_, out, _)
out[0..3] = 1
n := 1

Facts: \{val = 1\}
1. Start by sending $n = 1$

Ø (empty hypothesis)

=>

send(_, out, _)

out[0..3] = 1

n := 1

Facts: \{val = 1, out = &val\}

Show: (val = 1) \land (out = &val) \rightarrow (out[0..3] = 1)

Action: n := 1
3. Otherwise resend $n$

recv(_, in, _)
in[0..3] ≠ n
=>
send(_, out, _)
out[0..3] = n

Facts: {val = 1, n = 1, in = &recval, in[0..3] ≠ n}
3. Otherwise resend $n$

recv(_, in, _)
in[0..3] ≠ n
=>
send(_, out, _)
out[0..3] = n

**Facts:** {val = 1, n = 1, in = &recval, in[0..3] ≠ n, recval ≠ val}
3. Otherwise resend $n$

recv(_, in, _)
in[0..3] ≠ n
=>
send(_, out, _)
out[0..3] = n

**Facts:** {val = 1, n = 1, in = &recval, in[0..3] ≠ n, recval ≠ val, out = &val }

**Show:** out[0..3] = n
2. If $n$ is received, send $n + 1$

recv(_, in, _)
in[0..3] = n
$
\Rightarrow$
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: \{val = 1, n = 1, in = &recval, in[0..3] = n\}
2. If $n$ is received, send $n + 1$

recv(_, in, _)
in[0..3] = n
=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: \{val = 1, n = 1, in = &recval, in[0..3] = n, recval = val\}
2. If $n$ is received, send $n + 1$

```plaintext
recv(_, in, _)
in[0..3] = n
=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: \{val = 3, n = 1, in = &recval,
in[0..3] = n, recval = val\}
```
2. If $n$ is received, send $n + 1$

recv(_, in, _)
in[0..3] = n
=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: \{val = 3, n = 1, in = &recval, 
in[0..3] = n, recval = val, 
out = &val\}

Show: out[0..3] = in[0..3] + 1

Fails to verify!
How Much State to Keep?

• One option: Keep all knowledge of state

• Need to retain old information at assignment statements
  – \{val = 1, x = val\} val = 2; \{val = 2; x = val'; val' = 1\}

• Need to be *path-sensitive*
  – \{\} y = 1; if (p) then x = 1 else x = 2 \{ y=1; p=>(x=1); !p=>(x=2) \}

• These are both expensive!
Pistachio’s Design

• Maintain *must* facts
  – Subset of true facts; ones that definitely hold
  – Implies always safe to take subset

• Kill facts at assignments
  – \{val = 1, x = val\} val = 2; \{val = 2\}

• Intersect facts at join points
  – \{\} y = 1; if (p) then x = 1 else x = 2 \{ y = 1 \}

• Much more efficient
  – Loses precision
  – Aliasing issues cause some unsoundness
Fact substitution

recv(_, in, _)
in[0..3] = n
=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: \{val = 1, n = 1, in = &recval, in[0..3] = n, recval = val, \}

val is killed in the next statement

Using substitution, recval will still have a value of 1
Fixpoint Example

recv(_, in, _)
in[0..3] = n

=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: \{val = 2, n = out[0..3],
in = &recval, in[0..3] = n,
recval = 1, out = &val,
n = val, n = 2\}
Fixpoint Example

recv(_, in, _)  
in[0..3] = n  
=>  
send(_, out, _)  
out[0..3] = in[0..3] + 1  
n := out[0..3]

Facts: \{in = &recval, in[0..3] = n, n = val\}

We start the loop again with the intersection of the sets of facts from the first two iterations.
Fixpoint Example

recv(_, in, _)

\[ \text{in}[0..3] = n \]

=>

send(_, out, _)

\[ \text{out}[0..3] = \text{in}[0..3] + 1 \]

\[ n := \text{out}[0..3] \]

Facts: \{ in = &recval, \text{in}[0..3] = n, \n
\[ n = \text{val}, \text{recval} = \text{val} \}\}
Fixpoint Example

recv(_, in, _)
in[0..3] = n
=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: \{in = &recval, in[0..3] = n, val = n + 1, recval = val\}
Fixpoint Example

recv(_, in, _)
in[0..3] = n
=>
send(_, out, _)
out[0..3] = in[0..3] + 1
n := out[0..3]

Facts: {in = &recval, in[0..3] = n,
val = n + 1, recval = val,
out = &val}

Show: out[0..3] = in[0..3] + 1
Challenges

• Fixpoint may not converge
  – Gives up after 75 iterations
• For indirect assignments, only derive facts if write within bounds
  – And kill facts about the array otherwise
  – ...but do not forget everything else
• Functions inlined
• C data modeled as byte arrays
• Assume everything initialized to 0
Implementation

• Approximately 6,000 lines of OCaml
  – Uses CIL (http://manju.cs.berkeley.edu/cil/) to parse C programs
  – And Darwin as a theorem prover (http://combination.cs.uiowa.edu/Darwin/)
• Pistachio also uses user-provided specifications of library functions
  – In the same rule-based notation
Experimental Framework

• We used Pistachio on two protocols:
  – LSH implementation of SSH2 (0.1.3 – 2.0.1)
    • 87 rules initially
    • Added 9 more to target specific bugs
  – OpenSSH (1.0p1 - 2.0.1)
    • Same specification as above
  – RCP implementation in Cygwin (0.5.4 – 1.3.2)
    • 51 rules initially
    • Added 7 more to target specific bugs
• Rule development time – approx. 7 hours
Example SSH2 Rule

“It is STRONGLY RECOMMENDED that the ‘none’ authentication method not be supported.”

recv(_, in, _)
in[0] = SSH_MSG_USERAUTH_REQUEST
isOpen[in[1..4]] = 1
in[21..25] = “none”
=>
send(_, out, _)
out[0] = SSH_MSG_USERAUTH_FAILURE

If we get an auth request

For the none method

Then send failure
Example Bug

1. `fmsgrecv(clisock, SSH2_MSG_SIZE);`
2. `if(!parse_message(MSGTYPE_USERAUTHREQ, inmsg, len(inmsg),
    &authreq))`
3. `return;`
4. `if(authreq.method == USERAUTH_PKI) {
        .........
    }`
5. `} else if (authreq.method == USERAUTH_PASSWD) {
        .........
    }`
6. `} else {
        .........
    }`
7. `}
8. `sz = pack_message(MSGTYPE_REQSUCCESS, payload, outmsg,
    SSH2_MSG_SIZE);`
9. `fmsgsend(clisock, outmsg, sz);`

Message received
Handle PKI auth method
Handle passwd auth method
Oops – allow any other method
Send success; not supposed to send for *none* auth method
Another SSH2 Rule

“The server MUST respond to a TCP/IP forwarding request with the \textit{wantreply} flag set to 1 and the port set to 0 with a request success message containing the forwarding port.”

\begin{verbatim}
recv(in_sock, in, _ )
in[0] = SSH_MSG_GLOBAL_REQUEST
in[1..14] = “tcpip-forward”
in[15] = 1
in[(len(in) - 4)..(len(in) - 1)] = 0
=> send(out_sock, out, _ )
in_sock = out_sock
out[0] = SSH_MSG_REQUEST_SUCCESS
\end{verbatim}
Example Buffer Overflow Bug

0. char laddr[17]; int lport;

1. fmsgrecv(clisock,inmsg, SSH2_MSG_SIZE);
2. if(!parse_message(MSGTYPE_GLOBALREQ, inmsg, len(inmsg), &globalreq))
3.     return;

4. if(globalreq.msgtype==MSGTYPE_SUBTYPE_TCPIPFORWARD)
   
5.     strcpy(laddr,getstrfield(globalreq,payload,0));
6.     lport=getuint32field(globalreq,payload,1);

7. if(!create_forwarding(clisock,laddr,lport))
8.     return debug_error();
9. if(globalreq.wantreply==1) && (lport == 0)) {
10.    payload.msgid=SSH_REQUEST_SUCCESS;
11.    payload.reason=lport;
12.   sz = pack_message(MSGTYPE_REQSUCCESS, payload, outmsg, SSH2_MSG_SUZE);
13.   msgsend(clisock,outmsg,sz);

Watch this buffer
Receive message
If it’s a forwarding req
Uh-oh: strcpy to fixed buf (getstrfield may return >17 bytes)
Pistachio thinks this may fail, hence no msg sent
Causes of False Positives (LSH)
Discussion

• Network protocol implementations are a great target
  – Detailed specification available
  – Relatively small amount of code
  – Multiple implementations of the same protocol

• Better measurements of the utility of this analysis?
  – Able to find bugs that developers care about
  – How important were they?

• Could we eliminate these bugs in some other way?
  – A new language for network protocols?
  – What if used Pistachio during development?
Summary

- Rule-based specification closely related to RFCs and similar documents
- Initial experiments show Pistachio is a valuable tool
  - Very fast (under 1 minute)
  - Detects many security related errors
  - ...with low false positive and negative rates
Extra slides
Example library function rule

out = calloc(nelems, elsize)
nelems > 0
elsize > 0
=>
out ≠ NULL
out[0..(nelems * elsize – 1)] = 0
Legend

- Message structure and data transfer
  - Rules related to format, structure, and restrictions on messages
- Compatibility
  - Backwards compatibility with earlier protocol versions
- Functionality
  - What abilities should or should not be supported
- Protocol logic
  - Proper response sent for each message (these are the bulk of the rules)
Legend

- Interface
  - Errors in passing parameters
- Data access and handling
  - Indirect memory accesses, e.g., invalid pointers, out-of-bounds writes
- Data definition and declaration
  - E.g., forgetting to initialize variable, wrong data types
- Processing
  - General data-related computation errors
- Control flow and sequencing
  - Bugs related to branches and loops

** Note: bug categorization is an inexact process
Theorem Prover Choice

• True theorems = over all experiments
• Pistachio uses Darwin
  – Fastest theorem prover on average