

ProVerif

安装

ProVerif组件依赖

- OCaml
- graphviz
- (可选) LablGTK2
- (可选) GTK+2.24

如果需要使用ProVerif的交互式模拟功能，则需要安装上述的两个可选组件

下列安装过程基于ubuntu 18.04 LTS，windows安装参考[1]的1.4.3节

OCaml

```
1 | sudo apt-get install ocaml
2 | sudo apt-get install opam
```

graphviz

```
1 | sudo apt install graphviz
```

(可选) 安装LablGTK2与GTK+2.24

暂时用不到，略

有需要的话可以参考[2]的安装教程

安装ProVerif

到官网下载源码文件: <http://proverif.inria.fr/>

proverif是proverif2.05.tar.gz这个文件，proverifdoc2.05.tar.gz的是文档与相关文件的压缩包，建议两个一起下载，后者也有用

解压

```
1 | tar -xzf proverif2.05.tar.gz
```

进入对应目录，安装proverif

```
1 | cd ./proverif2.05
2 | ./build
```

这里如果没有安装上述两个可选组件的话，build会报面这个错误，但是实际上proverif是build成功的，只是无法使用交互式模拟功能

```
1 | ./build: 137: ocamlfind: not found
```

这里用 `-help` 参数可以看到proverif是可用的

```
snow@ubuntu:~/Documents/proverif/proverif2.05$ ./proverif -help
Proverif 2.05. Cryptographic protocol verifier, by Bruno Blanchet, Vincent Cheval, and Marc Sylvestre
-test                display a bit more information for debugging
-in <format>         choose the input format (horn, horntype, spass, pi, pitype)
-out <format>         choose the output format (solve, spass)
-o <filename>        choose the output file name (for spass output)
-lib <filename>       choose the library file (for pitype front-end only)
-set <param> <value> equivalent to adding set <param> = <value> to the input file
-TulaFale <version>  indicate the version of TulaFale when ProVerif is used inside TulaFale
-graph              create the trace graph from the dot file in the directory specified
-commandLineGraph   Define the command for the creation of the graph trace from the dot file
-gc                 display gc statistics
-parse-only          just parse the input file and report errors (if any)
-html               HTML display
-help              Display this list of options
--help             Display this list of options
```

握手协议

```
1  (* Symmetric Enc *)
2
3  type key.
4
5  fun senc(bitstring,key) : bitstring.
6
7  reduc forall m:bitstring,k:key;sdec(senc(m,k),k) = m.
8
9
10
11 (* Asymmetric Enc Structure *)
12
13
14 type skey.
15 type pkey.
16
17 fun pk(skey): pkey.
18
19 fun aenc(bitstring,pkey): bitstring.
20
21 reduc forall m:bitstring,k:skey;adec(aenc(m,pk(k)),k) = m.
22
23
24 (* Digital Signatures Structure *)
25
26 type sskey.
27 type spkey.
28
29 fun spk(sskey): spkey.
30 fun sign(bitstring,sskey) :bitstring.
31
32 reduc forall m:bitstring,k:sskey;getmess(sign(m,k)) = m.
33 reduc forall m:bitstring,k:sskey;checksign(sign(m,k),spk(k)) = m.
34
35
```

```

36
37 (* HandShake Protocol *)
38
39 free c:channel.
40
41 free s:bitstring [private].
42 query attacker(s).
43
44 let clientA(pkA:pkey,skA:skey,pkB:spkey) =
45   out(c,pkA);
46   in(c,x:bitstring);
47   let y = adec(x,skA) in
48   let (=pkB,k:key) = checksign(y,pkB) in
49   out(c,senc(s,k)).
50
51 let serverB(pkB:spkey,skB:sskey) =
52   in(c,pkX:pkey);
53   new k:key;
54   out(c,aenc(sign((pkB,k),skB),pkX));
55   in(c,x:bitstring);
56   let z = sdec(x,k) in
57   0.
58
59 process
60   new skA:skey;
61   new skB:sskey;
62   let pkA = pk(skA) in out(c,pkA);
63   let pkB = spk(skB) in out(c,pkB);
64   ( (!clientA(pkA,skA,pkB)) | (!serverB(pkB,skB)) )

```

协议分析

这里首先分析一下上面的代码的工作，proverif从process标记的主进程开始分析，所以这里看process后面的内容

首先是为clientA生成了非对称加密私钥skA，同时为serverB生成了签名用的私钥skB，并分别为这两个私钥生成对应的公钥（pkA，pkB），并通过公开信道c将这两个公钥发送出去

接下来调用两个进程宏clientA和serverB来实现两个进程的并发执行（Line 64），这里用replication的方式让两个主体都以无限数量会话并发执行（Line 64的clientA和serverB前面的感叹号表示重复无限多个会话并发）

然后关注clientA和serverB这两个模块，这里要把两个模块结合起来看，首先是A把自己公钥pkA发出去（Line 45），B会接收到这个公钥pkX（Line 52），同时创建一个对称密钥k（Line 53，k这里起到临时会话密钥的作用）

然后B将这个临时密钥k和自己的公钥pkB一起打包成元组，先用自己的私钥skB签名（Line 54的sign部分），再用收到的公钥pkX加密，并通过公共信道发出去（Line 54的aenc部分，这里aenc表示非对称加密）

A通过信道接收到这个发来的bitstring（Line 46），然后用自己私钥解密，解密成功之后得到y（Line 47），之后用B的公钥pkB验证签名，如果验证通过了就会得到元组(m,k)，第一项就是pkB（Line 48，这里用=pkB来匹配），第二项就是对称密钥k

A再用这个k给消息s进行对称加密然后发出去（Line 49），B接收到发来的bitstring之后，用自己刚刚创建的对称密钥k解密就得到了内容z，这应该和s是一样的内容

不难看出，这段代码很简单，所以也不难想到中间人攻击

proverif分析协议

把上述代码保存为 `handshake.pv` 文件，然后执行命令，可以得到proverif分析上述握手协议的结果

```
1 | ./proverif ./handshake.pv
```

ProVerif会输出其考虑过程的内部表示，然后一次处理逻辑中的查询（由query关键字表示），查询攻击者的输出可以分为三个部分

- Abbreviations到detailed部分：proverif推导出的导致攻击的流程
- detailed到trace has been found：描述了上述攻击的轨迹
- result：最终结论，true表示不存在攻击，false表示存在攻击者获取相关信息

这里分部分来看，首先是abbreviations的部分

```
1  -- Process 1 (that is, process 0, with let moved downwards):
2  {1}new skA: skey;
3  {2}new skB: skey;
4  {3}let pkA: pkey = pk(skA) in
5  {4}out(c, pkA);
6  {5}let pkB: spkey = spk(skB) in
7  {6}out(c, pkB);
8  (
9    {7}!
10   {9}out(c, pkA);
11   {10}in(c, x: bitstring);
12   {8}let skA_1: skey = skA in
13   {11}let y: bitstring = adec(x,skA_1) in
14   {12}let (=pkB,k: key) = checksign(y,pkB) in
15   {13}out(c, senc(s,k))
16 ) | (
17   {14}!
18   {16}in(c, pkX: pkey);
19   {17}new k_1: key;
20   {15}let skB_1: skey = skB in
21   {18}out(c, aenc(sign((pkB,k_1),skB_1),pkX));
22   {19}in(c, x_1: bitstring);
23   {20}let z: bitstring = sdec(x_1,k_1) in
24   0
25 )
26
27 Abbreviations:
28 k_2 = k_1[pkX = pk(k_3),!1 = @sid]
29
30 1. The attacker has some term k_3.
31 attacker(k_3).
32
33 2. By 1, the attacker may know k_3.
34 Using the function pk the attacker may obtain pk(k_3).
35 attacker(pk(k_3)).
36
37 3. The message pk(k_3) that the attacker may have by 2 may be received at
38 input {16}.
39 So the message aenc(sign((spk(skB[]),k_2),skB[]),pk(k_3)) may be sent to the
40 attacker at output {18}.
41 attacker(aenc(sign((spk(skB[]),k_2),skB[]),pk(k_3))).
```

```

41 4. By 3, the attacker may know aenc(sign((spk(skb[]),k_2),skb[]),pk(k_3)).
42 By 1, the attacker may know k_3.
43 Using the function adec the attacker may obtain
44 sign((spk(skb[]),k_2),skb[]).
45 attacker(sign((spk(skb[]),k_2),skb[])).
46
47 5. By 4, the attacker may know sign((spk(skb[]),k_2),skb[]).
48 Using the function getmess the attacker may obtain (spk(skb[]),k_2).
49 attacker((spk(skb[]),k_2)).
50
51 6. By 5, the attacker may know (spk(skb[]),k_2).
52 Using the function 2-proj-2-tuple the attacker may obtain k_2.
53 attacker(k_2).
54
55 7. The message pk(ska[]) may be sent to the attacker at output {4}.
56 attacker(pk(ska[])).
57
58 8. By 4, the attacker may know sign((spk(skb[]),k_2),skb[]).
59 By 7, the attacker may know pk(ska[]).
60 Using the function aenc the attacker may obtain
61 aenc(sign((spk(skb[]),k_2),skb[]),pk(ska[])).
62 attacker(aenc(sign((spk(skb[]),k_2),skb[]),pk(ska[]))).
63
64 9. The message aenc(sign((spk(skb[]),k_2),skb[]),pk(ska[])) that the
65 attacker may have by 8 may be received at input {10}.
66 So the message senc(s[],k_2) may be sent to the attacker at output {13}.
67 attacker(senc(s[],k_2)).
68
69 10. By 9, the attacker may know senc(s[],k_2).
70 By 6, the attacker may know k_2.
71 Using the function sdec the attacker may obtain s[].
72 attacker(s[]).
73
74 11. By 10, attacker(s[]).
75 The goal is reached, represented in the following fact:
76 attacker(s[]).

```

1. attacker用一个已知的私钥 `k_3` 生成对应的公钥 `pk(k_3)` (对应abbreviations的前两步, 这里的 `k_3` 可以是attacker自己生成的, 或者通过其他方式已知的)
2. 然后这里标记了代码的{16}位置, 该位置表示可以从信道中收到攻击者发出的公钥 `pk(k_3)`, 同时在{18}位置会将消息 `aenc(sign((spk(skb[]),k_2),skb[]),pk(k_3))` 消息通过信道发出, 此时attacker可以接收到这条消息 (也即attacker获取到了这个知识)
3. 然后attacker可以从信道中获得这条加密的消息 (`aenc`), attacker利用私钥 `k_3` 对其解密, 可以得到 `sign((spk(skb[],k_2),skb[]))`, 并通过 `getmess` 获取签名的内容 (也即 `(spk(skb[]),k_2)`), 这里意味着attacker可以获得 `k_2`
4. 之后attacker获取A的公钥 `pkA`, 并用该公钥对 `sign((spk(skb[]),k_2),skb[])` 进行加密并发送出去
5. 这条消息会在{10}被client收到, client验证签名正确之后, 会在{13}将消息 `senc(s[],k_2)` 发送出去, attacker会收到这条消息
6. 之后attacker就可以用 `k_2` 解密得到 `s[]`

然后看detailed output的部分

```

1 new skA: skey creating skA_2 at {1}
2

```

```

3  new skB: sskey creating skB_2 at {2}
4
5  out(c, ~M) with ~M = pk(skA_2) at {4}
6
7  out(c, ~M_1) with ~M_1 = spk(skB_2) at {6}
8
9  out(c, ~M_2) with ~M_2 = pk(skA_2) at {9} in copy a
10
11 in(c, pk(a_1)) at {16} in copy a_2
12
13 new k_1: key creating k_4 at {17} in copy a_2
14
15 out(c, ~M_3) with ~M_3 = aenc(sign((spk(skB_2),k_4),skB_2),pk(a_1)) at {18}
    in copy a_2
16
17 in(c, aenc(adec(~M_3,a_1),~M)) with aenc(adec(~M_3,a_1),~M) =
    aenc(sign((spk(skB_2),k_4),skB_2),pk(skA_2)) at {10} in copy a
18
19 out(c, ~M_4) with ~M_4 = senc(s,k_4) at {13} in copy a
20
21 The attacker has the message sdec(~M_4,2-proj-2-
    tuple(getmess(adec(~M_3,a_1)))) = s.
22 A trace has been found.

```

- 前两行对应的是公钥密钥对和签名密钥对的创建过程
- 然后是三个out，前两个out表示attacker将加密公钥和签名公钥分别保存在 ~M 和 ~M_1 中，第三个out表示client在{9}处的输出，attacker将其保存为 ~M_2
- 从Line 9开始，之后的每一行后面都跟了一个 in copy a 或者 in copy a_2，相同的标识表示相同的会话，copy a 表示当前attacker正在与client进行通信，copy a_2 表示当前attacker正在和server通信
- 这里看Line 11-15，表示attacker向server发送了一个公钥，同时server返回了一个用server私钥签名并加密的消息，这里attacker将这条消息保存为 ~M_3 (Line 15)
- 然后在看Line 17-19，这里表示attacker在{10}这里，用 a_1 这个私钥解密 ~M_3，再用client的公钥加密（这里的 ~M 就是client的公钥），将加密后的消息发送出去，当client在{13}返回加密的消息时，attacker可以利用 k_4 解密并获得 s

这里可以回顾中间人攻击的流程，攻击者需要同时维护两个会话（对应于上面的两个会话），将从与server会话接收到的内容选择性的篡改（也可以不改，上面的协议需要解密，所以修改了），然后发送到与client的会话中

由于没有额外的方式完成鉴权（上述过程只有一个server的签名和认证），所以client和server都认为它们与对方建立了加密会话，而实际上这个加密会话所使用的密钥已经被attacker获取到了

下面放一张图，更清晰一些

