Modeling Replay And Integrity Violations Attacks For Cryptographic Protocols Source Codes Verification Of E-Voting System Based On Blind Intermediaries

Institute of Computer Technologies and Information Security. Southern Federal University

Professor, Doctor of Technical Sciences, Liudmila Babenko, lkbabenko@sfedu.ru

Graduate student, Ilya Pisarev, ilua.pisar@gmail.com

Classic development

1. Protocol creation

2. Description on Alice-Bob language like: A->B: Ek(m)

3. Description on verificator's specification language

4. Verification by special tools like Avispa, Scyther, ProVerif, Tamarin Prover.

5. Protocol repairing if attacks were founded and go to step 3.

6. If protocol is secure – implementation it on programming language for secure system

Problems

- Details lost due to abstract protocol form

Limitations and assumptions of protocols verifiers. For example, symmetric encryption modes, integrity control are not considered.

- Importance description on difficult specification languages

Number of verifiers have difficult language to describe behavior of protocol (like HLPSL in Avispa) and it can take a lot of time to describe it qualitatively and correctly.

- Doesn't guaranty protocol secure implementation (MAIN PROBLEM)

Protocol can be modified during implementation on program language, it can be additional messaging between the parties, it can be protocol-logic errors on source code like lack of checking the returned random number in request-response schemes, it can be simple human factor that can lead to any other error in the source code.

Implementation problems

Alice-Bob form of previous protocol:
1. A -> B: E_kb(Na, A)
2. B -> A: E_ka(Na, Nb)

!=

Alice-Bob form of implemented protocol:
1. A -> B: E_kb(M1L)
2. A -> B: E_kb(Na, A)
3. B -> A: E_ka(M2L)
4. B -> A: E_ka(Na, Nb)

Protocol in theory:
1. A -> B: E_k(Na)
2. B -> A: E_k(Na, kNew)
If *Na* in message 1 not equal *Na* in message 2 so break connection because someone intervened in connection.

!=

Protocol practice implementation:
1. A -> B: E_k(Na)
2. B -> A: E_k(Na, kNew)
No checks for *Na* in source code due to programmer error what mean that protocol not working at practice.

Source code to Alice-Bob format

Source code on programming language->	AnB (Alice-Bob) format

…
byte[] M1enc;
using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
{
  RSA.ImportParameters(rsaPB.ExportParameters(false));
M1enc = RSA.Encrypt(M1, true);
}
 	
socA.Send(M1enc);
…
	1. A -> B: E_key(ID, Password)
	2. B -> A: E_key(Answer)

Previous source code part

```
…
//Some other systems methods

void SomeMethod()
{
  IPEndPoint remoteEP =
new IPEndPoint(SomeSystemsIP, SomeSystemsPort);
  …
  byte[] ID =
Encoding.Unicode.GetBytes(textBox1.Text);
  byte[] password =
Encoding.Unicode.GetBytes(textBox2.Text);

  byte[] M1 = Enc(ID.Concat(password).ToArray(),
key);

  soc.Send(M1)

  byte[] M2 = new byte[64];
  soc.Receive(M2)
  …
}

…
//Some other systems methods
```

Reduced system source code part

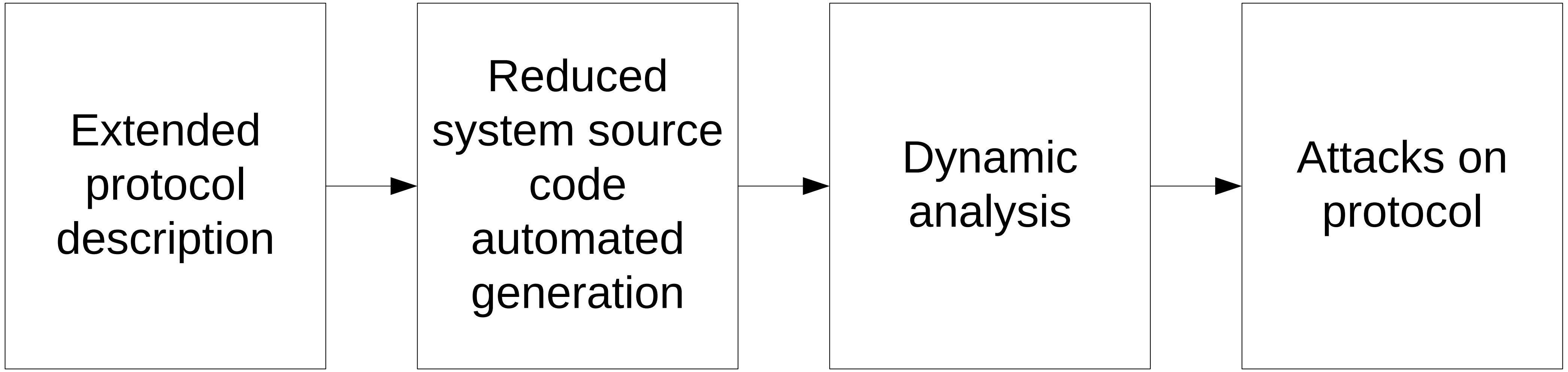
```
void SomeMethod()
{
  IPEndPoint remoteEP = new IPEndPoint(LocalIP,
LocalPort1);
  …
  byte[] ID =  File.ReadAllBytes("file1.txt");
  byte[] password =
File.ReadAllBytes("file2.txt");

  byte[] M1 = Enc(ID.Concat(password).ToArray(),
key);

  soc.Send(M1)

  byte[] M2 = new byte[64];
  soc.Receive(M2)
  …
  File.WriteAllLines(@"ClientFLAG.txt",
new string[] { "Good" });
}
```

Dynamic analysis scheme


Extended protocol description
Reduced system source code automated generation
Dynamic analysis
Attacks on protocol

Dynamic analysis. principle of “false termination”.

Client:	Channel	Server:
1. Ek(M1)->	->	1. ->Ek(M1)
2. Ek(M2)<-	<-	2. <-Ek(M2)
3. Ek(M3)->	->M3->	3. ->M3
“Correct end flag”		“Correct end flag”

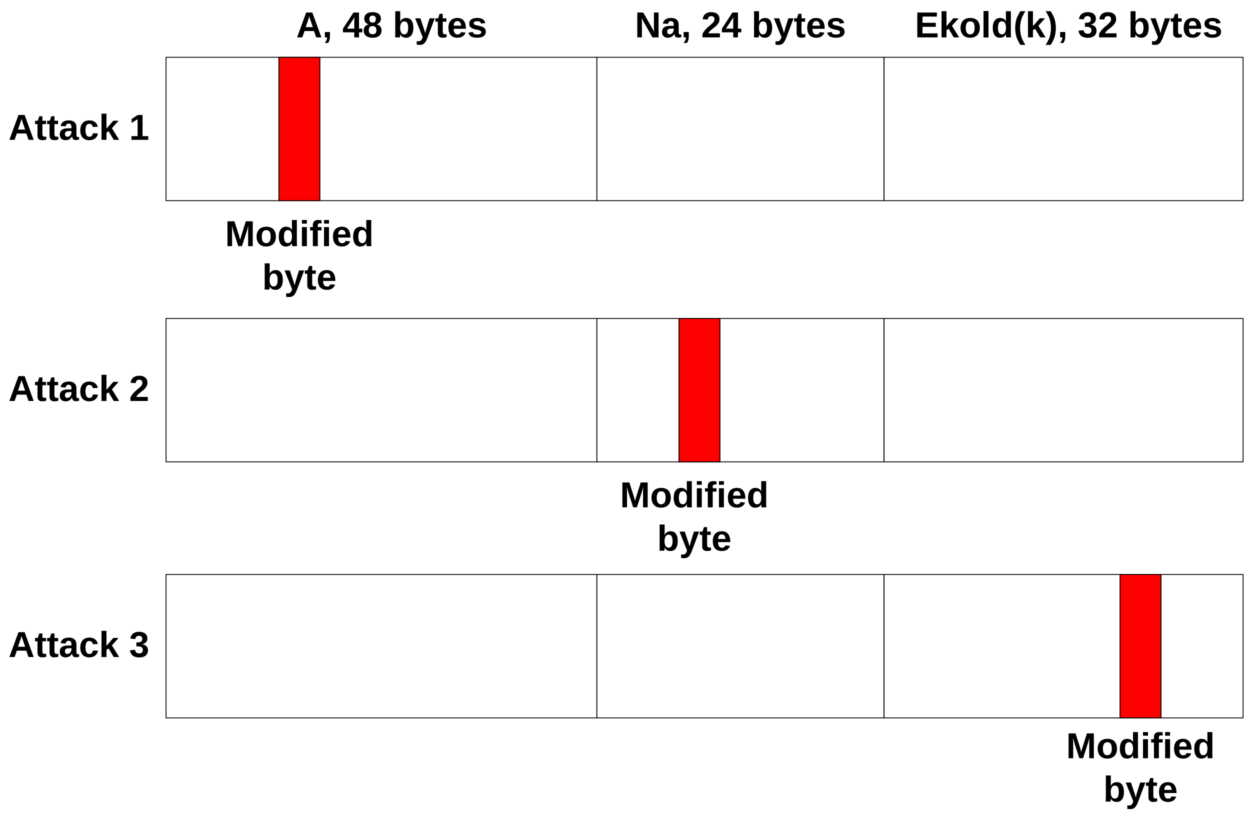
If “Correct end flag” on all sides – an attack was found

Replay attack examples

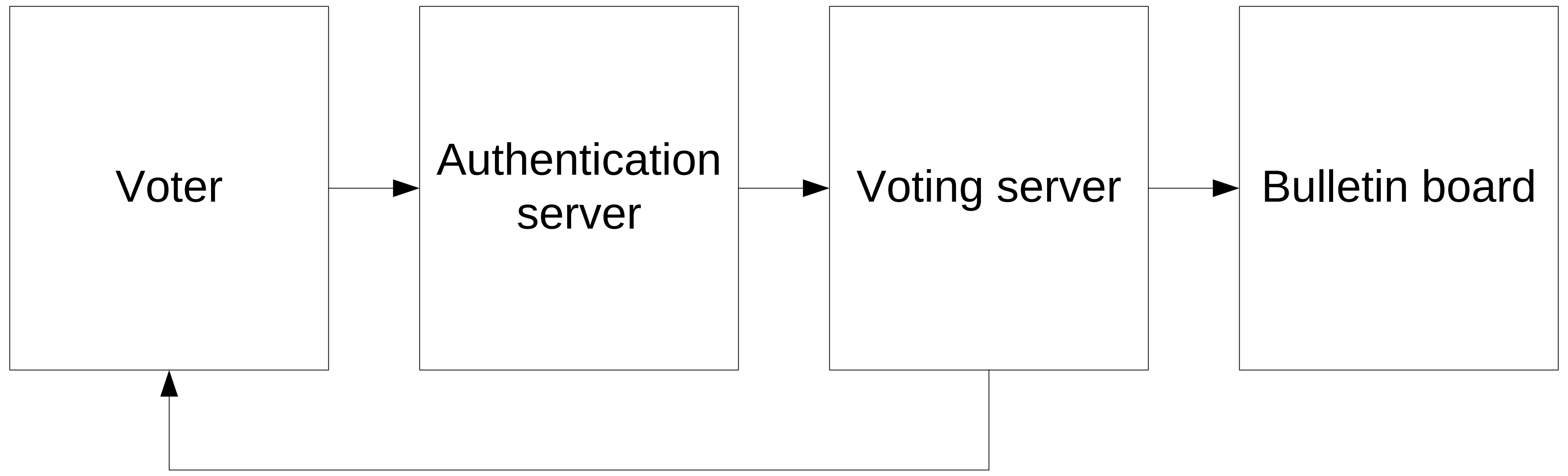
Attack 1	**Attack 2**	**Attack 3**
1. A->B: M1	1. A->B: M1	1. A->B: M1
2. B->I: M2	2. B->A: M2	2. B->A: M2
I->A: M1	3. A->I: M3	3. A->I: M3
3. A->B: M3	I->B: M1	I->B: M2

Attack 1	**Attack 2**	**Attack 3**	**Attack 4**
1. A->I: M1	1. A->I: M1	1. A->B: M1	1. A->B: M1
I->A: M1_old	I->B: M2_old	2. B->I: M2	2. B->I: M2
2. B->A: M2	2. B->A: M2	I->A: M1_old	I->A: M2_old

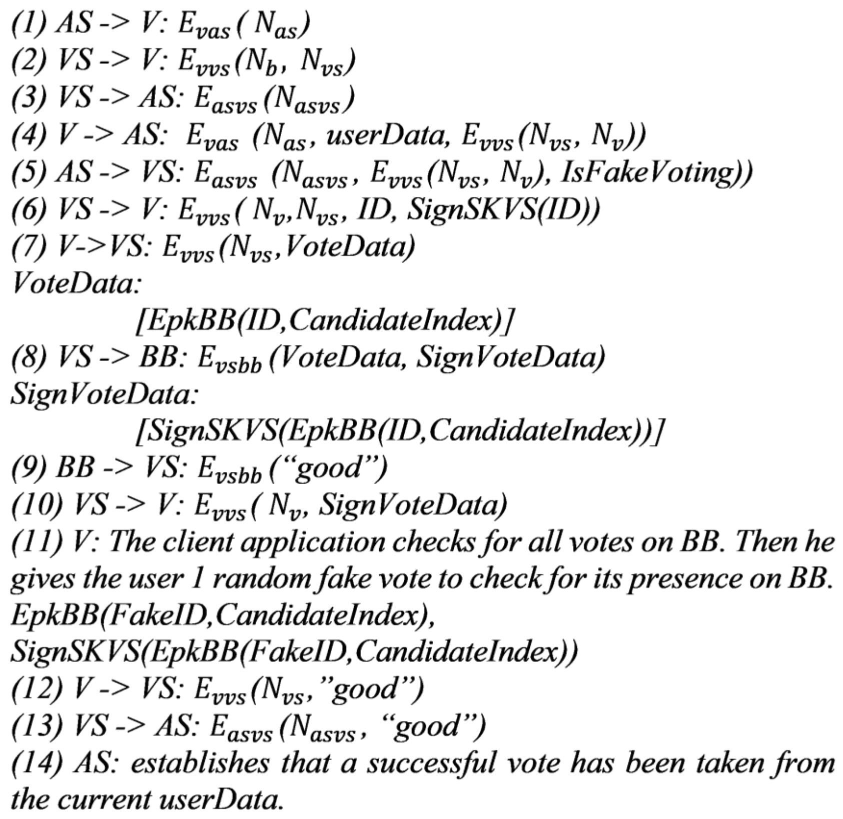
Integrity violation attack examples



E-vote system based on blinded intermediaries



E-voting protocol



(1) AS -> V: E_{vas}(N_{as})
(2) VS -> V: E_{vvs}(N_b, N_{vs})
(3) VS -> AS: E_{asvs}(N_{asvs})
(4) V -> AS: E_{vas} (N_{as}, userData, E_{vvs}(N_{vs}, N_v))
(5) AS -> VS: E_{asvs} (N_{asvs}, E_{vvs}(N_{vs}, N_v), IsFakeVoting))
(6) VS -> V: E_{vvs}(N_v,N_{vs}, ID, SignSKVS(ID))
(7) V->VS: E_{vvs}(N_{vs},VoteData)
VoteData:
[EpkBB(ID,CandidateIndex)]
(8) VS -> BB: E_{vsbb}(VoteData, SignVoteData)
SignVoteData:
[SignSKVS(EpkBB(ID,CandidateIndex))]
(9) BB -> VS: E_{vsbb}(“good”)
(10) VS -> V: E_{vvs}(N_v, SignVoteData)
(11) V: The client application checks for all votes on BB. Then he gives the user 1 random fake vote to check for its presence on BB.
EpkBB(FakeID,CandidateIndex),
SignSKVS(EpkBB(FakeID,CandidateIndex))
(12) V -> VS: E_{vvs}(N_{vs},”good”)
(13) VS -> AS: E_{asvs}(N_{asvs}, “good”)
(14) AS: establishes that a successful vote has been taken from the current userData.

Dynamic analysis results.

On implementation all messages sends by this construction:

(1) AS -> V: $E_{vas}(MessL1)$

(2) AS -> V: $E_{vas}(Message)$

Results. Replay attack

(1) AS -> V: E_{vas}(MessL1)
(2) AS -> V: E_{vas}(N_{as})
(3) VS -> V: E_{vvs}(MessL2)
(4) VS -> V: E_{vvs}(N_b, N_{vs})
(5) VS -> I -> AS: E_{asvs}(MessL3)
I: E_{asvs}(MessL3)
(6) VS -> I -> AS: E_{asvs}(N_{asvs})
I: E_{asvs}(N_{asvs})
(7) V -> AS: E_{vas} (MessL4)
(8) V -> AS: E_{vas} (N_{as}, userData, E_{vvs}(N_{vs}, N_v))
(9) AS -> VS: E_{asvs} (MessL5)
(10) AS -> VS: E_{asvs} (N_{asvs}, E_{vvs}(N_{vs}, N_v), IsFakeVoting))
(11) VS -> V: E_{vvs}(MessL6)
(12) VS -> V: E_{vvs}(N_v,N_{vs}, ID, SignSKVS(ID))
(13) V->VS: E_{vvs}(MessL7)
(14) V->VS: E_{vvs}(N_{vs}, VoteData)
(15) VS -> BB: E_{vsbb}(MessL8)
(16) VS -> BB: E_{vsbb}(VoteData, SignVoteData)
(17) BB -> VS: E_{vsbb}(MessL9)
(18) BB -> VS: E_{vsbb}("good")
(19) VS -> V: E_{vvs}(MessL10)
(20) VS -> V: E_{vvs}(N_v, SignVoteData)
(21) V -> VS: E_{vvs}(MessL11)
(22) V -> VS: E_{vvs}(N_{vs},"good")
(23) VS -> I: E_{asvs}(MessL12)
I -> AS: E_{asvs}(MessL3)
(24) VS -> I: E_{asvs}(N_{asvs}, "good")
I -> AS: E_{asvs}(N_{asvs})

Weakness description and how to fix

There was a check for “good” in the source code, but no action was taken on this check. In all other checks in the code, when the returned values do not match, the protocol ends, and this event is written to the log. This attack itself does not affect the quality of this protocol, but this flaw can cause a desync error. This message determines whether the authentication server installs information that the user has successfully voted. Due to this flaw, an intruder can affect the security of the voting procedure, since it can block the data transmission channel between the voter and the voting server, and then send a confirmation message to the authentication server. As a result, the authentication server will assume that the user has voted correctly, but his vote on the voting server will not be counted due to the blocking of the channel. To correct this defect, it is necessary to add the correct processing if the return value does not match, by analogy with other places in the source code. In this case, the implementation of the protocol will be completely protected from replay-attacks and integrity violations.

Future work. DPA language. Custom modeling.

```
Projects
{
  Info1: “project_client”, “ProtocolCall(bool mode)”,
“123”, “project_client/protocol.cs”
  Info2: “project_server”, “ProtocolCall(bool mode)” ,
“223”, “project_server/protocol.cs”
}

Parameters
{
  A: Info1
  B: Info2
  I: INTRUDER
}

MainProtocolsConnection
{
  1 A=>B
}
MainProtocol1 //Please use this as base for models
{
  A=>B
  1 [A = “sertA.dat”, Na = RAND, kb = “keyB.dat”] A-
>B: E_kB(Na, A)
  2 [B = “sertB.dat”, Nb = RAND, ka = “keyA.dat”, Na
= Get_1_Na] B->A: E_kA(Na, Nb, B)
  3 [kb = “keyB.dat”, Nb = Get_2_Nb] A->B: E_kB(Nb)
}
```

```
Model1
{
  A=>I
  I=>B
  1 [A = “sertA.dat”, Na = RAND, kI =
“keyI.dat”] A->I: E_kI(Na, A)
  2 [A = Get_1_A, Na = Get_1_Na, kb =
“keyB.dat”] I->B: E_kB(Na,A)
  3 [B = “sertB.dat”, Nb = RAND, kA =
“keyA.dat”, Na = Get_2_Na] B->A: E_kA(Na,
Nb, B)
  4 [kI = “keyI.dat”, Nb = Get_3_Nb] A->I:
E_kI(Nb)
  5 [kb = “keyB.dat”, Nb = Get_4_Nb] I->B:
E_kB(Nb)
}

Run
{
  Model1
}
```

Conclusions

The paper describes a dynamic analysis method for detecting replay-attacks and integrity violations attacks. The proposed approach allows security verification of cryptographic protocols at the last stage of their development, which is the most effective way to detect attacks. Dynamic analysis was applied to readymade e-voting system, which developed on C# language. We found replay-attack. This attack can lead to components desynchronization. So, voting server think that voter doesn’t vote, but authentication server think that voter vote. Protocol itself is secure, but there was an implementation error. In source code part with last message was no correct check processing. After repairing this code error this protocol implementation will be secure from replay-attacks and integrity violation attacks. The future direction of work is to add security and authentication attacks support, as well as increase the variations of simulated attacks.