WAR: Wireless Anonymous Routing

Michael E. Locasto
Clayton Chen
Ajay Nambi
Department of Computer Science
Fu Foundation of Engineering and Applied Science
Columbia University
{locasto, nambi}@cs.columbia.edu
ccc57@columbia.edu

Abstract

Wireless Anonymous Routing (WAR) is a proposed mechanism for defeating Traffic Analysis (TA) in a wireless environment. The current approach consists of three protocols based on mechanisms used to frustrate TA attempts in a wired (fixed network) setting. These mechanisms are onion routing and traffic mixing with cover traffic. Refinements to these protocols will be explored as part of future work. The first protocol under consideration will be implemented and tested in both a stand-alone and a simulation environment. Performance metrics as well as the efficacy of the anonymity mechanisms will be recorded and analyzed.

The protocol presented in this paper continues this theme by describing the mechanisms necessary for ensuring that traffic analysis attempts in a wireless environment are frustrated. The implementation of this protocol in a network of wireless devices is a crucial step toward preserving privacy and confidentiality in a mobile ad-hoc network.

Keywords

wireless routing, mobile ad-hoc network, traffic analysis, onion routing, wireless security, wireless anonymity

1. Related Work

The primary focus of current research in anonymous routing is the discovery and application of methods to make anonymous both the connection and the actual communication. However, most of this work is being done in fixed (wired) networks. Routing in traditional networks is subject to a number of attacks; the wireless environment is even more exposed. The wireless arena has many more
constraints and problems to be solved; actually assuring communication at all is a difficult problem! It is no surprise that there is little work being done to secure or disguise traffic or routing in a wireless environment because it adds another layer of complexity in an already intricate environment.

The literature presented here focuses on Onion Routing, the WAR protocols as envisioned by John Ioannidis, Matt Blaze, and Angelos Keromytis, and current work in mobile ad hoc networks (MANETs).

1.1 Onion Routing

Onion Routing was conceived in 1996 by David M. Goldschlag, Michael G. Reed, and Paul F. Syverson for the NRL's research group in high assurance systems. Onion routing is an architecture in which a proxy uses special routers to create anonymous connections to other proxies[8]. The proxy creates a message that contains several layers of encryption (an onion) corresponding to each router along the
connection path. Each router removes a layer of encryption and forwards the result along the path.

In addition to onion routing, the protocols employ cover traffic and traffic mixing. These are the well-known technique of including spurious cover traffic to even out the total frequency distribution of traffic on the network as well as hide timing and size correlation between packets on the network.

1.2 Approach

The WAR protocols are based on a combination of onion routing and traffic mixing with cover traffic. A primary problem in the wireless arena (as well as other places) is distribution of the keys used to encrypt traffic. In WAR, the keys are distributed with each RadioGram. However, a full PKI is beyond the scope of the current problem.

It is particularly easy to perform Traffic Analysis (TA) in a wireless setting because traffic is by nature broadcast into the "airnet." [3] Also as easy is linking physical location to identity; simple triangulation on a transmission source provides the point of origin and identity can be established any number of ways (source address, unique physical properties of transmission device, visual, etc).

Our approach is based on a RadioGram object [1,2] that is flooded across the wireless network. The RadioGram object contains several layers of encrypted data that effectively encapsulates the actual content. A RadioGram consists of an RSA public key and a structured byte array that is encrypted several times by the transmitter. Each "skin" of the encryption corresponds to a node in the intended success path of the RadioGram (a hop in the route). The byte array contains enough redundancy to guarantee that a node can recognize that it has properly decrypted a skin of the RadioGram. We plan to incorporate the exposition of various performance metrics into the simulation framework for later data analysis.

1.3 Design

The design of the protocol is based around the processing of a "RadioGram" object, which is a specially formatted byte array containing some content and the attendant information that provides onion routing. A radiogram is defined by a transmitter identification (the RSA public key of the transmitter, preceded by an unencrypted length) and a content byte array. The byte array consists of an RSA encrypted session key preceded by an unencrypted length, a 128 bit message integrity code (MIC), 4 bytes of control signals, genuine content, and random padding.

[tID] {[sessionkey][MIC][^]}{[sk.i][MIC.i][^.i]}{...}[content][padding]

1.3.1 KEY & TRANSMITTER ID

We assume it is the RSA Public Key of the transmitter.
This is how someone becomes known on the network. The transmitter ID is a series of bytes representing the RSA public key. This ID is used to encrypt the session key. This hybrid mode encryption provides greater performance. The session key is then used to encrypt the rest of the content array, including the control signals and MIC.

1.3.2 CONTENT

The content is broken down into five primary components: MIC, SESSION KEY, CONTROL SIGNALS,

CONTENT, and PADDING.

1.3.3 CHECKSUM or MIC

The MIC is 128 bits or 16 bytes long. It is a pre-computed hash value of everything the current onion skin wraps, except the padding. A specific and detailed specification is given below in the examples. It is encrypted under the session key of the current onion skin (the session key that immediately follows it).

1.3.4 SESSION KEY

This portion of the content array is a symmetric key encrypted by the public key of the transmitter. The initial design uses an AES key, which has 128 bits (16 bytes), however, the symmetric encryption mechanism can be any strong secret-key algorithm. If another algorithm (e.g., DES) is chosen, then the rest of the content array should be offset the required amount. In addition, since the RSA encryption produces a non-predictable length result, some bytes are prepended to the session key to indicate its encrypted length.

1.3.5 CONTROL SIGNALS

The signals are the most complex arrangement in the content array. The signals tell the receiver what to do with the received message. The signals are four bytes wide. Two bytes are dedicated to control, and two bytes are dedicated to indicating how much padding is currently appended to the message.

1.3.6 CONTENT

This portion of the array is the actual content to be passed. The content may have additional authentication tokens embedded in it. Only the final destination (indicated by a control signal of type MESSAGE) will be able to interpret the content according to the application-level requirements. The content, for any given layer of an onion, may actually be encrypted content under another node's RSA public key.

1.3.7 PADDING

The portion of the content array that is random nonsense. Since the padding changes in transit for a given message, the attacker can never be sure that simply disregarding the last bits or assuming they are
random will gain him anything. In addition, the RadioGram will tend to grow smaller over time as it follows the "success path." Therefore, a potential attacker could potentially analyze most of any given route for a RadioGram. Padding the message to one of several predefined sizes or allowing for random expansion and contraction guards against the
attacker assuming that a RadioGram's size provides any information about the stage of the RadioGram's lifecycle.

1.4 CONTROL SIGNAL SPECIFICATIONS

The first byte is dedicated to specifying both predefined and vendor-specific control signals. In the protocol, there are four pre-defined control signals (each is shown with hex, decimal and binary equivalent):

- COVER (0x10, 16, 0001 0000)
- MESSAGE (0x60, 96, 0110 0000)
- ENCAPSULATED (0xB0, 176, 1011 0000)
- CONTROL (0xF?, ??, 1111 yyyy)

There are three pre-defined CONTROL control signals:

CONTROL_GEN_COVER (0xF0, 240, 1111 0000)
CONTROL_ADD_PAD (0xFF, 255, 1111 1111)
CONTROL_SUB_PAD (0xFE, 254, 1111 1110)

The remainder of the 1111 yyyy series are vendor specific control codes. Any control signal below 1111 0000 is reserved for future modifications to the protocol. The second byte is for vendor specific purposes, and is reserved for use in the standard protocol only when the first byte is 1111 1110 or 1111 1111. In the latter case, the second byte is then used to indicate the amount of padding (in bytes) to be added or subtracted at that stage of the RadioGram's life. The second byte is NOT used by the protocol if the first byte is 1111 0000, however, it may be used by the vendor if the first byte is 1111 0000.

The third and fourth bytes indicate how much total padding (in bytes) is actually appended to the message. With 16 bits in these two bytes, there can effectively be 2^16 or 65536 bytes of padding present.

1.5 PACKET FORMAT

The packet format of the radiogram is constructed according to the following layout:

[tID] [session key][MIC] [^] [content][padding]
[z bits][m bits] [128 bits][32 bits][n bits] [k bits]
{ may be repeated (onion skin) }

The size of any given RadioGram is a function of how many times it has been wrapped, the size of its content, and the current size of its padding.

SIZE = 1024+S*(128+128+32)+C+P

where SIZE is the total logical size of the RadioGram in bits, S is the number of onion skins, C is the content length, and P is the padding length. So, a message wrapped twice with content length of 2048 bits and 32 bits of padding will be:

SIZE = 1024+2*(288)+2048+32
SIZE = 3680 bits or 460 bytes

The size of the content clearly dominates the equation. Padding can be pre-calculated to disguise a message or change its size during transit. This allows both random sized messages and predetermined sized messages (small, medium, large).

1.6 EXAMPLES:

If Alice wishes to send a message to Dave via Bob and Charles, the resulting packet would undergo the following transformations by Alice:

A --content--> D via B and C

Alice needs Bob's, Charles's, and Dave's public keys.
Alice needs to randomly select three 128 bit AES session keys and encrypt them under Bob's, Charles's, and Dave's public keys. Alice

needs to determine how much size delta the message will undergo during transit and adjust MIC and control signals appropriately. Alice needs to compute a MIC for each stage of the onion and prepend the MIC.
Alice computes the MIC using the unencrypted session key, over the current control signals and the encrypted remainder of the message (including other encrypted session keys, MICs, control signals, and content. Current padding is not included. The MIC is computed as a keyed hash using the current decrypted session key.

At each stage, the intended recipient obtains the 2nd 128 bits of the content array and attempts to decrypt it with its private key. This operation should produce an AES session key. The AES session key is then used to decrypt the remainder of the content array and verify the MIC. If the MIC
does not match, then the message is not meant for the current recipient.

So, Alice builds the following structure:

```
[A.id][B.sk][B.MIC][B.^][C.sk][C.MIC][C.^][D.sk][D.MIC][D.^][content][pa
d]
    {.....encrypted with B's session key.....}
                {.....encrypted with C's session key....}
                            {......encrypted with D.sk..}
```

by performing these steps:

1. Generate content [content]
2. Generate a random 16 byte array, call it D.sk
3. Set the control signals D.^ to type MESSAGE
4. Set the control signals D.^ to k bits of padding
5. Prepend [D.^] to [content]
6. Compute a 16 byte MIC using MD5 over [D.sk][D.^][content], call it D.MIC
7. Encrypt [D.MIC][D.^][content] under D.sk
7. Encrypt D.sk with D's public key, call it D.sk'
8. Prepend [D.sk'] to [D.MIC][D.^][content]
10. Append any needed padding. (optional step)
11. Rename [D.MIC][D.sk'][D.^][content][padding] to [content]
12. Goto step 2, repeat for C and B.

The only tricky part is how to manage and compute the bits of padding. Alice must instruct Bob and Charles how much padding to ignore in their computation of the current MIC. This of course includes any padding that Alice instructed the previous recipient to append.

If a node other than Bob gets this packet, that node will drop it once it sees that the bits corresponding to B.MIC do not match its computed MIC. These are the failure paths of the packet. Our concern with them stops here.

If Bob receives this RadioGram, Bob attempts to interpret the RadioGram as follows:

1. Bob strips A.id and saves it if he did not know it.
2. Bob strips B.MIC and saves it
3. Bob locates and strips the encrypted B.sk'
4. Bob decrypts B.sk' to B.sk with his private key
5. Bob uses B.sk to decrypt the remainder of the message
6. Bob assumes the message has integrity and that it is for him
7. Bob looks at the "decrypted" B.^ to find where padding begins
8. Bob chops off padding as indicated by B.^
9. Bob computes B.MIC' over [B.sk][B.^][content]
10. Bob compares B.MIC' and B.MIC
11. If they are equal, Bob's assumption is correct. Bob looks into B.^ for more information. If they are unequal, Bob's assumption is incorrect. The RadioGram is either not for Bob or has been altered in transit. Bob drops or logs the packet as required by the application domain.
12. If B.^ indicates type MESSAGE, Bob reads the content (passes the RadioGram up his protocol stack). If B.^ indicates type ENCAPSULATED, Bob takes any other actions specified by B.^, strips B.^, prepends his transmitter ID, and puts the packet in his outgoing queue.

If B.^ indicates type COVER, Bob can retransmit this message. It is a meaningless message. Bob can also drop it. If B.^ indicates type CONTROL, Bob takes the control signal action.

13. Bob consults his random oracle and sees if the time has elapsed for generating random COVER traffic. If so, he generates some amount and places it in his outgoing queue.

Step 10 is crucial. If step 3 has succeeded (the session key was actually encrypted with Bob's public key) and step 6 has succeeded, Bob does not actually know they have succeeded until step 9 and 10, where he can compare the MICs. Until then, he is under the assumption (read delusion) the RadioGram is for him. Alice may also have instructed Bob to add enough padding so that the loss of B.MIC, B.^, and B.sk' do not indicate that a message has been successfully routed further on. On the other hand, Alice may tell Bob to
remove some random padding to play with an attacker's head, or add enough padding to get the RadioGram back to a 'standard' size (small, medium, or large).

The RadioGram now looks like this:

```
[B.id][C.MIC][C.sk][C.^][D.MIC][D.sk][D.^][content][pad]
    {.............encrypted with C's session key......}
                {.......encrypted with D.sk.}
```

Charles repeats this procedure. Again, if any other node receives this message, it is a failure path and the node will discover this when it attempts to decrypt the packet and compare MICs.

Then, Charles sends out a packet that looks like this:

```
[C.id][D.MIC][D.sk][D.^][content][padding]
    {.....encrypted with D.sk...}
```

David receives this message. Any other node is a failure path and takes the previously described action.

David decrypts and checks the MIC and control signals. The type will be of type MESSAGE, so David will read the message and use its information, after dropping the padding as specified by the unencrypted [D.^].

2. Implementation Discussion

A detailed explanation of the current implementation

is presented. Next, several drawbacks to the current design are discussed. Finally, future work is addressed.

2.1 Implementation

The first WAR protocol is implemented in Java, using the 1.4.1 SDK. The WAR protocol is implemented in two primary threads and packaged with a simple API for reuse in any client program. A simple chat client with a Swing GUI was developed to showcase the WAR protocol in action.

A standalone thread, **edu.columbia.cs.war.WarThread**, was written to accept some configuration information and begin listening on a multicast channel. Once **WarThread** obtains a connection, it creates a new instance of **edu.columbia.cs.war.Worker**. **Worker** then processes the received bytes according to the WAR protocol described above. **Worker** may post either a cover message or an encapsulated message to its outgoing queue. The class **edu.columbia.cs.war.Sender** runs periodically and empties the queue, sending whatever radiograms are present to the multicast group. The class **edu.columbia.cs.war.McastOnion** contains the facility to create a layered onion. Both **McastOnion** and **Worker** call the cryptographic routines provided in the **edu.columbia.cs.war** package; the implementation uses the RSA and AES algorithms provided by the BouncyCastle encryption service provider for Java 1.4.x.

The **WarThread** has a simple API to manage its lifecycle, from configuration, through initialization, to cleanup and shutdown. The **WarThread** is easy to configure; a client simply provides some well-known attributes and calls two methods.

2.2 Current Problems

The current design of the protocol has several drawbacks. The primary difficulty is key distribution; however, since mechanisms exist to solve this problem, it is not addressed. However, since this design makes use of a public key infrastructure, the time needed to deliver a packet from the origin to the destination is considerably long due to RSA encryption and decryption. Although improvements have been made by creating a session key for encrypting the message contents and have the session key RSA encrypted before delivery, the layers made to create the onion will certainly increase the delivery time, especially if the participant hardware is not adequate to perform fast computations of decryption.

To encrypt a "layer", there are basically three expensive operations:

-generate a "random" session key
-encrypt message w/ symmetric session key
-encrypt session key w/ RSA key

The cost of the first two actions are negligible, especially compared with the third action. There are two approaches that may be taken to reduce the cost of RSA encryption during packet formation.

In order to improve the delivery time, one can use the session key twice and save the time required for the RSA encryption. Currently each packet will contain a new session key for encrypting the packet content. To improve the delivery time, the session key can be used for a second time, thereby skipping the RSA encryption procedure. Also, the nodes that are responsible for "peeling the onion" can save extra time by caching the last RSA decrypted session key. This approach can reduce the amount of time taken to create and deliver a packet by half while preventing the message from being compromised, especially if the AES cipher is used.

However, key reuse is not an optimal strategy. Instead, it may be possible to amortize the cost of creating RSA encrypted session keys for the hybrid-mode encryption. If the sender (any node in the network or ad-hoc group) maintains a cache or pool of 3-tuples {nodeID, P(session key), C(session key) } where nodeID is the identifier for a peer node in the group, P(session key) is a randomly generated AES session key stored in "plaintext", and C(session key) is the RSA-encrypted version of the stored session key, using nodeID's public key.

Using the pool or cache would remove the generation of an RSA-encrypted session key from the critical path of packet creation. This approach represents a significant performance gain. The pool would be refreshed and garbage collected by a background thread.

2.3 Joining the multicast group

The current design also doesn't specify how each user would participate in or split from the conversation. Currently, each end user has all the participating information of the group. If one decides to leave the multicast session, other end users would either need to be notified or have a keep-alive timer for each participant. Similarly, a new participant is not known by others when she joins the multicast group.

A proposed solution is to have the participant regularly send out her public key information. However, this does not prevent any intruders from joining the session and pretend to be one of the participants. In a wireless environment, there is no formal join-leave protocol at the physical or link layers. A timeout or formal "goodbye" protocol would have to be established at a higher layer, however, due to the unpredictable nature of the wireless environment, nodes may not be able to participate in such a "goodbye" protocol. Thus, the timeout approach makes more sense.

2.4 Delivery of packets

The protocol doesn't provide the guarantee for packet delivery. The current design will allow end users to pick her specific path to deliver her message while the participants would not leave the session. That is, she can pick the users who would participate for delivering the packet to the final destination. However, the packet cannot be delivered if a participant suddenly quits. If a single point of link fails, the packet will never reach its final destination. Also, as mentioned above, since the participants cannot tell if any participant has left or not, other users may pick the delivery path involving those who have already left the session. A solution would have the users send multiple packets with different paths of delivery while suffering from the increase of congestion of the network. But this tradeoff may not justify for increasing the probability of delivery, especially this functionality is not the current design's focus.

3. Results

The protocol proof of concept and development environment was implemented using the Java 1.4.1 platform. Performance metrics include:

- encryption time per layer
- packet creation time
- decryption time for one layer (1 peel attempt)
- measuring delays, including those due to network congestion, sending thread sleep time, and display polling.

Preliminary testing of the system reveals that the system works well for three nodes; performance is acceptable and any lag is noticeable, but not frustrating. It takes between 1 and 3 seconds for the message to be delivered, depending on the path and the variable factors listed above.

Three important results are the realization of additional requirements for survivable networks of wireless anonymous router systems: the need to incorporate key distribution mechanisms, the need to recognize and prevent "failure chaining", and the need to detect Byzantine failure of packet delivery and peer disconnection.

4. Conclusions and Future Work

The WAR protocol is a good general solution to the problem of anonymous routing in a wireless environment. Successive transformations of the protocol for performance and security are anticipated. Most notably, the performance enhancement of calculating RSA encrypted session keys offline will be implemented.

5. References

[1] WAR Protocols (protocol1.txt, protocol2.txt, protocol3.txt) as distributed by JI.

[2] DESIGN (the design document for protocol 1)
http://www.cs.columbia.edu/~locasto/projects/war/docs/DESIGN

[3] Summary of Secure Wireless Routing Issues
http://www.isi.edu/workshop/public_html/wmcw97/tsudik.html

[4] Wireless Routing Simulation Software
http://sourceforge.net/projects/wrss/

[5] Onion Routing Project
http://www.onion-router.net/

[6] Paul F. Syverson, Michael G. Reed, and David M. Goldschlag, "Onion Routing Access Configurations," DISCEX 2000: Proceedings of the DARPA Information Survivability Conference and Exposition, Volume I Hilton Head, SC, IEEE CS Press, January 2000, pp. 34--40.

[7] Paul F. Syverson, Gene Tsudik, Michael G. Reed and Carl E. Landwehr, "Towards an Analysis of Onion Routing Security," Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, July 2000

[8] Michael G. Reed and Paul F. Syverson, "Onion Routing," Proceeding of AIPA '99, March 1999.

[9] RFC 2501 http://www.ietf.org/rfc/rfc2501.txt

[10] The Zone Routing Protocol (ZRP)
http://www.ietf.org/internet-drafts/draft-ietf-manet-zone-zrp-04.txt