# Reader Authentication Protocol for EPCglobal C1G2 CRFID Tags

Jonathan Blieschke

***Abstract*—With Computational RFID tags being put to use in wireless sensor networks, protocols are being developed which allow a sensor's firmware to be updated wirelessly in deeply-embedded applications. So far, little work has been done to enable secure updates, such that only authorized readers can install firmware onto tags. We have implemented a reader authentication protocol for the WISP5 CRFID tag. While the protocol we have implemented is not completely secure, we have proposed modifications and extensions which improve the security while still allowing it to be implemented on top of the WISP5's relatively limited firmware. Continued work in this area could bring about a cryptographically secure wireless firmware update protocol.**

## I. Introduction

THIS project began with an investigation into the security measures available on a Wireless Identification and Sensing Platform revision 5 (WISP5) CRFID tag.[1] Recognising that RFID tags are becoming more commonplace for inventory purposes, and the potential uses for CRFID devices in wireless sensor networks, we originally hoped to find and exploit a vulnerability in the WISP5 firmware, and possibly implement a fix. We began with an investigation into the EPCglobal protocol version 1.2,[2] which we believed to be current at the time when the WISP5 firmware was developed. We examined the protocol specification, looking for ways in which a reader could interact with a tag which might cause the tag to behave unexpectedly, if care was not taken when writing the firmware.

Initially, we identified a number of commands with variable lengths, as these could have been vulnerable to a buffer overflow. We also noticed that commands such as *BlockWrite*, if not bounds-checked, could possibly be used to read or write areas of memory which are not supposed to be accessible. The usefulness of such a vulnerability would depend on the layout of the tag's memory, and the range of accessible locations. Shortly after we began looking through the WISP5 code, we were quite disappointed. No variable-length commands were correctly implemented, and the parameters read from *Read*, *Write* and *BlockWrite* commands were stored in a globally-accessible structure in the tag's memory rather than being handled in accordance with the protocol specification. When each of the commands were completed, they would call 'hook' functions, which are intended to be implemented by other developers to extend the WISP5's behaviour. The write hook, for instance, could be used to decide whether the data sent by the reader should be stored in the tag's non-volatile memory, and if so, where. Since the default implementations of these hooks is a single no-op, we realised that any vulnerabilities we found would need to be in applications built on top of the firmware.

While searching for academic papers which had implemented useful applications on the WISP5, two papers on wireless firmware updates caught our attention. [3], [4] Both of these papers acknowledged that they lacked any form of protection, allowing attackers to plant their own firmware on the tag, and recommended that this be a focus of future work. So, shifting our focus slightly away from exploiting vulnerabilities, we turned our attention to the other part of our goal and attempted to secure these wireless update protocols. We decided against recommending the *Access* command described in the EPCglobal version 1.2 specification, as it is trivial to steal the tag's access password by eavesdropping on its exchange with the reader.

## II. Previous Work

### A. Over-the-Air Firmware Update

The $R^2$[3] and Wisent[4] papers both present similar mechanisms for transmitting new firmware to a tag, utilising the *Write* command's MemBank field to differentiate between different reprogramming stages. Both protocols use a MemBank of $00_2$ to begin the reprogramming process, $01_2$ to write a piece of the new firmware, and $10_2$ to end the reprogramming process, executing the new firmware. Performing regular writes to the tag's data memory is done with a MemBank of $11_2$.

Both protocols also outline a bootloader which allows multiple 'applications' to be installed on a tag, and for different applications to be selected and run. Both papers cite Bootie,[5] an earlier proposal for a CRFID bootloader which aimed to achieve the same goal. One noticeable difference from Bootie is that both of these papers use the execution command to select which application is run, allowing a reader to execute one arbitrarily with a single command.

The two most significant security risks on tags which implement these protocols are the installation and execution of firmware by an unauthorised reader. In order to effectively secure this protocol, a reader must be prevented from beginning the reprogramming process and from executing an installed application unless it has been authenticated. Security measures which are implemented at the EPC protocol level - such as the *Access* command - would be suitable, but are not implemented on the WISP5.

### B. EPCglobal Protocol Version 2

In 2015, EPCglobal released an update to the C1G2 protocol.[6] Significant extensions were included as optional features, many of which directly addressed the security concerns in the older protocol. Several new commands were added, including*Challenge*, *Authenticate*, *SecureComm*, *AuthComm*, *KeyUpdate* and *TagPrivilege*.

The *Authenticate* command is an alternative to *Access*, which was present in the previous protocol version. Like *Access*, *Authenticate* causes a tag to transition from the open state into the secured state. Unlike *Access*, it uses one of the cryptographic suites present on the tag to verify the reader, the tag, or both. The cryptographic suites available on a particular tag are defined by the manufacturer, and a parameter passed to the *Authenticate* command specifies which suite is to be used. Some suites may require multiple commands in order to authenticate successfully. *Challenge* is used to instruct multiple tags to prepare for a subsequent authentication. As with *Authenticate*, one of its parameters specifies which cryptographic suite should be used.

A tag may contain a number of cryptographic keys, with each key having an individually assigned list of privileges. When a reader authenticates with a particular key, the commands which it may send to a tag depend upon the privileges associated with the key it used. The *KeyUpdate* command is used to modify keys, if the key used to authenticate has appropriate privileges. The *TagPrivilege* command is used to modify the privileges granted to keys, if the key used to authenticate already has appropriate privileges. *AuthComm* and *SecureComm* are used to securely encapsulate other commands, in a manner specified by the cryptographic suite.

These protocol extensions outline a general security framework for CRFID tags and their data, including the possibility of role-based authorization. Implementing a firmware update protocol on a tag which supports mutual cryptographic authentication would eliminate the existing security concerns. Since the framework outlined in this updated specification is an industry standard, it provides a solid foundation for designing an authentication protocol suited to a more specific purpose.

## III. Reader Authentication Protocol

### A. Description

Our first idea for securing the wireless update protocols was to improve the security of the EPCglobal 1.2 access procedure. The reader begins by sending a *ReqRN* command to the tag, which backscatters a random number. The reader uses this number to cover-code the 16 most-significant bits of the 32-bit access password, which it sends back to the tag. The reader then issues another *ReqRN*, and the process repeats for the 16 least-significant bits of the access password. If both halves of the password are received correctly, the tag transitions into the secure state. The most obvious problem with this procedure is that an eavesdropper could easily steal the password. By overhearing both random numbers and both cover-coded halves of the password, the attacker has enough information to uncover the entire password, compromising the security of the tag. This issue could be solved relatively simply, by having the tag encrypt the random number before sending it. Thus, when the tag receives a correctly cover-coded password, it can be certain that the reader also shares its encryption key, and the password cannot be stolen by eavesdropping. While this does appear to fix the problem, any tag which implements this solution exactly as stated is no longer compliant with the EPCglobal specification. Implementing this solution on the WISP5 would also be challenging, since significant amounts of time-sensitive assembly code would need to be modified. An easier solution would be to implement a similar system as an application on top of the WISP5 firmware. Some care would need to be taken to ensure that the implementation remains compatible with the wireless update protocols, but small adjustments could be made to either protocol in the interests of improving their security.

To begin our implementation of an access procedure, we need to be able to encode the *ReqRN* and *Access* commands within other commands, such as *Read* or *Write*. First, the tag must be instructed to generate and encrypt a random number. Borrowing an idea from the wireless update protocols, we set aside a particular *Write* command to initiate the authentication procedure. Since the WISP5 does not implement password protection, use of the reserved memory bank ($00_2$) for access-specific commands seems appropriate. Performing a *Write* to the reserved memory bank with WordPtr set to zero will cause the tag to generate a new random number, encrypt it, and store the encrypted value in the memory bank at WordPtr = 4. This requires some slight modification of the WISP5's default layout, as its reserved bank usually resides in volatile memory. However, modifying the pointer in the relevant global structure resolves this problem. The EPCglobal specification does not specifically prohibit the inclusion of other data in the reserved memory bank, so this protocol remains compatible. The data sent to the tag via this write command is ignored. At the next stage, the reader must read the encrypted random number out of the tag's memory. Issuing a *Read* command to the reserved bank with WordPtr = 4 will retrieve the data stored in the memory bank. The random number itself will only be 16 bits long, but more data may need to be read depending on which cipher was used. Our implementation used AES, a 128-bit block cipher, requiring that we read the entire ciphertext from the tag. The *Read* command allows up to 255 words (4,080 bits) to be read from a single command, so this is easily achieved. The reader will decrypt the received random number, and use it to cover-code a 16-bit password, much like the one used in the EPCglobal access procedure. The reader will then send this cover-coded password back to the tag. It should be sent via a *Write* command to the tag's reserved bank, with WordPtr $> 0$. The tag will check that the received password is correct, thus verifying that the reader it is communicating with has both its password and its cryptographic key. Thus,

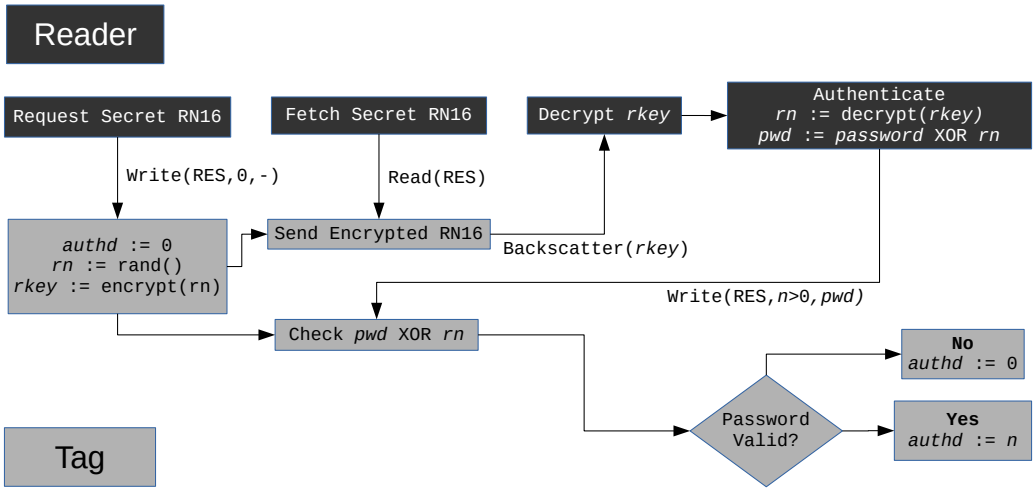


Fig. 1. Flowchart of the Authentication Protocol

the reader is considered authenticated until it instructs the tag to generate a new random number. Figure 1 illustrates the protocol we implemented.

A reader's authentication status only affects its ability to write to the tag's memory, and not its ability to read like the EPCglobal access procedure. Since this is sufficient to protect the wireless update protocols from malicious firmware, this protocol is still sufficient to address the most significant concerns identified in their respective papers. There are several reasons for this limitation, mostly related to simplifying the implementation. Writing data to the tag's memory is handled by hooks, which the tag's firmware calls after the command is processed, but the read hook is not called until the data has already been transmitted to the reader. Checking the tag's authentication status before responding to a read would require the modification of time-sensitive assembly code. The tag would also need to allow the encrypted random number to be read, but nothing else, which complicates the necessary checks.

### B. Possible Extensions

We considered some possible extensions to this protocol, such as having 'sessions' which expire after a specified number of writes. The value of WordPtr on the second *Write* would be used to set a counter, which is decremented whenever a write is performed. Written data is only stored in the tag memory if the counter is greater than zero. When the tag is instructed to generate a new random number, it resets the counter to zero. Expiring sessions may be useful for quickly writing a fixed amount of data to a tag, as it removes the need for an additional command to de-authenticate when the writes are complete. However, in a streaming application like a Wisent data transfer, a limit of 255 successive writes may simply be an inconvenience. The other possible protocol extension was inspired by the support for multiple keys and encryption methods in the updated EPCglobal protocol. The data sent by the first *Write* command to initialise the authentication procedure could be used to specify which key and/or encryption method to use, if more than one is supported. This could be extended further by only allowing some keys to write new applications onto the tag, while others are allowed to execute existing applications. Alternatively, this data field could be used to encode commands for a more complex authentication protocol. An example of such a protocol will be described later.

### C. Implementation

Our implementation used AES, a 128-bit block cipher, recommended by a colleague as one of the better algorithms implemented by the BLOC project.[7], [8] The 16-bit random number we generated was stored in the first two bytes of the encrypted block. The rest of the block was unspecified values, often zeroed out or populated with the value $FF3F_{16}$. We hard-coded the encryption key into the tag's firmware. We implemented the entire tag-side protocol within the write hook, deciding what to do primarily based on the MemBank field and an authentication flag outside of the tag's accessible memory. We also ensured that the unencrypted random number was not accessible via *Read* command. We implemented the session-counter extension, where each authenticated write would decrement the authentication counter. Unfortunately, our implementation mistakenly placed the reserved memory bank, unencrypted random number and authentication counter in volatile memory, preventing it from working when the tag is not connected to a debugger. We did not realise this error until we no longer had enough time to fix it.

### D. Problems & Possible Solutions

While a high-level description of this protocol seems relatively secure, there are still a number of problems. First, due to the limited computational resources available on a passive CRFID tag like the WISP5, having the tag calculate a random number is not ideal. The WISP5 specifically uses a lookup table, which makes the 'random' numbers it generates relatively predictable. The second problem is the length of the tag's password. The 32-bit password used by the EPCglobal

TABLE I
EXAMPLE OF ADDITIONAL COMMANDS EMBEDDED IN THE DATA FIELD

| | RFU | Sending | Data |
|---|---|---|---|
| Bits | 6 | 2 | 8 |
| Use | - | $00_2$: Reset Authentication<br>$01_2$: Random Number<br>$10_2$: Password<br>$11_2$: Set Expiration | Data |

access procedure has been criticised[9] for its vulnerability to a brute-force attack. The shorter 16-bit password used in this protocol is significantly easier to guess, which would also allow an attacker to bypass decrypting the random number. These two vulnerabilities are significant, but by no means irreparable. However, both solutions would benefit from a *BlockWrite* implementation which adheres to the EPCglobal specification. The WISP5 is designed to work with a reader which does not issue the *BlockWrite* command correctly. Instead of writing the block of data in a single command, the reader issues a series of *BlockWrites*, each one writing only a single word of data, making it effectively equivalent to an ordinary *Write*. These proposed solutions will initially assume a correctly-implemented *BlockWrite*, and then a more complex solution will follow for the WISP5.

To initiate the authentication procedure, the reader writes an encrypted random number into the tag's reserved memory bank, at WordPtr = 0. For ciphers like AES, with a block size larger than 16 bits, a *BlockWrite* is necessary to send the entire ciphertext in a single command. The tag resets its authentication state, decrypts the random number, and prepares to receive a cover-coded password. The encrypted random number is not saved to the reserved bank, but the decrypted number is stored in memory which cannot be accessed with a *Read* command. The reader then writes a cover-coded password to the tag's reserved memory bank at WordPtr $> 0$. The tag checks the password, and the reader is successfully authenticated. This protocol allows both major problems to be solved at once. Because the reader generates the random number, not the tag, much more computational power is available, so the number will be harder to guess than the existing solution. The use of *BlockWrite* also allows much longer random numbers and passwords to be used, significantly reducing the likelihood that an attacker can simply use brute-force to obtain access to the tag. The random number and password could both utilise the cipher's full block size, rather than being restricted to the first two bytes.

To implement an improved protocol on the WISP5, modifications must be made to remove the need for a *BlockWrite* command. We propose that these modifications could make use of the data field in the first command of our existing protocol. If the 16-bit data field of this command is used to encode other protocol-specific commands, such as beginning the transfer of a multiple-byte ciphertext. Table I illustrates a simple example of the behaviour which could be embedded in the data field. When a *Write* is received to the reserved memory bank with WordPtr = 0, the data field is not written to the tag's memory but is parsed as a protocol command. The first six bits are currently unused. The next two bits encode the command which the tag should parse. $00_2$ indicates that the tag should reset its current authentication status. $01_2$ indicates that the next series of writes to the reserved memory bank will be the encrypted random number. Each write in the series will use the WordPtr field to indicate the order in which they should be stored. $10_2$ indicates that the next series of writes to the reserved memory bank will be the cover-coded password. Writes will behave the same as for transmitting the random number. $11_2$ indicates that the tag's expiration counter should be set to the value stored in the data field. If the data field contains a zero, the reader's authentication will not expire until it is reset. While the implementation of this protocol would be challenging, it would allow tags to authenticate a reader before saving data to their memory banks. In order to allow compatibility with $R^2$ or Wisent, this protocol may require further modification, but this should provide a sold foundation for future work in this area.

## IV. FUTURE WORK

Despite the progress that we have made in this direction, more work yet needs to be done before wireless firmware update protocols like $R^2$ or Wisent can benefit from this reader authentication protocol. Conflicts in functionality between their reprogramming commands and the first command from this protocol need to be identified and resolved, then the combined secure protocol needs to be implemented. Further extensions to the combined protocol could allow for the implementation of a system which brings functionality similar to version 2.0 of the EPCglobal specification onto devices which only support an older version. Further analysis could also be done on which cryptographic cipher is most suitable for this particular application, as our implementation was based on a cursory reading of the BLOC project analysis paper.

## V. CONCLUSION

We have implemented a reader authentication protocol for the WISP5 CRFID tag. While the protocol we have implemented is not completely secure, we have proposed modifications and extensions which improve the security while still allowing it to be implemented on the WISP5's relatively limited firmware. Continued work in this area could bring about a cryptographically secure wireless firmware update protocol.

## REFERENCES

[1] https://wisp5.wikispaces.com/WISP+Home
[2] EPC$^{TM}$Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz - 960 MHz Version 1.2.0 (2008) http://www.gs1.org/sites/default/files/docs/epc/uhfc1g2_1_2_0-standard-20080511.pdf
[3] D. Wu, M. J. Hussain, S. Li and L. Lu, "R2: Over-the-air reprogramming on computational RFIDs," 2016 IEEE International Conference on RFID (RFID), Orlando, FL, USA, 2016, pp. 1-8. doi: 10.1109/RFID.2016.7488004

[4] Tan, Jethro, et al. "Wisent: Robust Downstream Communication and Storage for Computational RFIDs." arXiv preprint arXiv:1512.04602 (2015).

[5] Ransford, Benjamin. "A Rudimentary Bootloader for Computational RFIDs." University of Massachusetts Amherst, Tech. Rep. UM-CS-2010-061 (2010).

[6] EPC$^{TM}$Radio-Frequency Identity Protocols Generation-2 UHF RFID Specification for RFID Air Interface Protocol for Communications at 860 MHz - 960 MHz Version 2.0.1 Ratified (2015) http://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard.pdf

[7] Mickal Cazorla, Kevin Marquet, and Marine Minier. Survey and benchmark of lightweight block ciphers for wireless sensor networks. IACR Cryptology ePrint Archive, 2013:295, 2013. http://eprint.iacr.org/2013/295.pdf.

[8] Mickal Cazorla, Kevin Marquet, and Marine Minier. Survey and benchmark of lightweight block ciphers for wireless sensor networks. In Pierangela Samarati, editor, SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavk, Iceland, 29-31 July, 2013, pages 543-548. SciTePress, 2013.

[9] Contactless Smart Cards vs EPC Gen 2 RFID Tags: Frequently Asked Questions http://www.smartcardalliance.org/publications-epc-gen2-faq/