This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the ICC 2007 proceedings.

S.T.R.E.S.S. : Stress Testing and Reverse Engineering for System Security

Matteo Rosi, Leonardo Maccari, Romano Fantacci
Department of Electronics and Telecommunications - University of Florence
Telecommunication Network Lab
tel. : +390554796467 - fax : +390554796485 Florence, Italy
Email: {rosi, maccari, fantacci}@lart.det.unifi.it

***Abstract*—**
In modern wireless networks the functions included into layer II have to deal with complex problems, such as security and access control, that were previously demanded to upper layers. This growing complexity led some vendors to implement layer II primitives directly in software, e.g. IEEE 802.11i has been largely distributed as a software patch to be used with legacy 802.11b/g hardware. In any extremely complex software the likelihood of committing errors during the implementation raises, and it is well known that software bugs can lead to instability of the system and possibly to security vulnerability. Software bugs are the most common cause of successful attacks against any kind of network and represent a real plague for system administrators. Stress test is a widely used methodology to find and eliminate software bugs. In this paper we present a platform to perform a stress test of generic network protocols implementations but especially optimized for Layer II stress tests, that present specific problems. With our approach a generic network protocol described with ABNF language can be tested transmitting arbitrary frame sequences and interpreting the responses to verify consistence with the communication standard used. Our platform can interact dynamically with the tested machine (an access point, a router etc.) to verify its robustness and its compliance with the standard. Experiments confirmed the validity of our approach both as a stress test technique for system under development and as a reverse engineering technique for interaction with closed source system.

I. Introduction

With the enrichment of the functionalities included in layer II for modern wireless standards, the functions included into MAC Layer are growing in complexity. As an example in *IEEE 802.11i* [1] the MAC layer is responsible for authentication and cryptography, so that complex protocols such as TLS handshakes (Transport Layer Security) or WPA (Wireless Protected Access, defined in the standard) have to be pushed down in the protocol stack. Traditionally, MAC functions are implemented as firmware primitives in the physical interfaces or as low level software primitives included into the drivers. The growing complexity, and the continuous need to update to new standards led many vendor to move as much features as possible into software driver. As said, *IEEE 802.11i* standard is a security enhancement that can be partially applied to existing *IEEE 802.11* interfaces; several vendors decided to give driver updates to support this enhancement also on legacy hardware.

The more complex a software is, the higher is the possibility of committing errors during the implementation. An error, as explained in following sections, can lead to instability of the software or in worse cases can be exploited by an attacker to force the platform to perform actions controlled by the attacker himself. Moreover, well known attack techniques such as buffer overflow of format bugs that are commonly used in higher layer software now apply also to layer II primitives, so that exploits can be publicly available. For the outlined reasons, during driver development it is extremely important to minimize the number of errors, performing stress tests on the products and verifying that the responses given are consistent with the standard used. Performing a test basically means sending to the device under test a sequence of frames and verify its reactions; a stress test is a chain of tests containing also malformed data. As we will see later, stress tests for layer II software has some specific difficulties that we plan to address with our platform. These are mainly the direct interaction with low level drivers and the need to filter layer II frames that can be coming from any sufficiently close network.

In this article we present *S.T.R.E.S.S.*, a modular, dynamic platform for layer II software testing, that respects the following requisites:

- It must be generic and not bound to a specific standard. A key feature of *S.T.R.E.S.S.* is the possibility of interpreting any standard protocol described with a high level syntax.
- It must be modular, so that extending the use to new physical devices must be easy.
- It must be dynamic, the platform should not just blindly inject a packet sequence into the network, but it should reinterpret the responses and react accordingly.
- Tests must be easy to perform and easily repeatable with different devices using the same protocol.

Reverse engineering is the activity of reconstructing the behaviour of a software or hardware component whose source code is not public, with the aim of extending its compatibility. *S.T.R.E.S.S.* can be also used as a reverse engineering platform; during our experiments we successfully used it to understand why the application of a custom protocol (described in [2]) to a CISCO access point was not successful. *S.T.R.E.S.S.* can be used to verify that a certain device is standard compliant as it declares to be. Lastly, *S.T.R.E.S.S.* can be very useful for penetration testing and bug hunting activities.

1-4244-0353-7/07/$25.00 ©2007 IEEE

```
vect = malloc(10);
   // allocate a vector of 10 char
strcpy(vect, buffer);
   // copy into vect the value of buffer
printf("%s", vect);
   // print the content of vect
```

Fig. 1. Example C code containing a bug

In section II we will introduce testing methodologies and examine possible approaches, in section III we will describe *S.T.R.E.S.S.* in detail and in last section we will illustrate results of applications in a real scenario.

II. Stressing approach

Testing is the process of discovering errors present in software programs or hardware's firmware. This process is based on the definition of a *chain of events* made up of three states: error, fault and failure. An *error* in a software, commonly called *bug*, is a mistake in software development committed by the programmer. An error may not manifest itself every time a program is run, but only within particular conditions. Whenever the error shows off, it generates a *fault*. A fault is an incorrect condition of the program, for example, a wrong I/O operation on the memory or a method that returns a wrong result. A fault may generate a *failure*: a software produces a failure when it cannot accomplish its requisites, in other words it doesn't do what the user expects from it. With software Testing we can try to find errors in source code and to certify if the program under test is fail safe, that is, a fault never generates a failure.

As an example, consider the C code in fig. 1: If `buffer` is longer then `vect` the second line writes into a memory zone that is not allocated. This code can be considered an error, but it may be inserted in a program in which it never causes any faults (i.e. if the lenght of `vect` always equals the lenght of `buffer`). Otherwise, let's say `buffer` is longer than `vect`, this situations depicts a fault because the result of the action may be unpredictable (it is unclear what the `printf` will do). In the worst case, writing into an unchecked buffer may completely block the program, that will exit with a *segmentation fault* error, producing a *failure*.

There are two main types of software testing:

- Structural Testing (White-Box)
- Functional Testing (Black-Box)

Structural testing tries to obtain its goal using implementation details that must be available to the tester, i.e. source code. On the other hand, functional testing feeds the program with various inputs and analyzes the program output verifying that the requisites of the program are respected. Structural testing can be done on open source (whose source code it's publicly available) projects or during the development of the product while functional testing can be performed by anybody using the software, such as the committers of the software or security experts whose role is evaluating the software security.

Our goal is to develop a platform to evaluate security and standard compatibility of networking software and hardware solutions; since source code or other technical details about the implementation are rarely available, we choose a functional approach.

Stress testing is a specific type of functional testing, it is aimed at inducing the software to enter an incorrect state feeding it with malformed or uncommon inputs. The idea behind this technique is that software errors often reside in uncommon situations that are rarely handled by the software and less carefully reviewed by the developers. To find software errors we try to force the software to manage uncommon situations and verify its reactions. Basically, for testing we introduce into the communications some *faults*, that we create artificially, these faults are called *anomalies*. In network testing, the input to the application under test is a sequence of frames to be transmitted over the physical media and anomalies can be frames of a wrong type, or some incorrect value in a particular frame; injecting an anomaly we force the system to enter a wrong state, as said, we artificially introduce a fault. We expect the tested application to handle anomalies in a correct way, reporting an error according to the standard procedures withouth falling into a failure.

Testing applied to layer II software presents some peculiar difficulties, expecially with a wireless media:

- for a complete test of a wireless protocol we must communicate to a low layer in ISO-OSI stack, so we have to interact with a non *human readable* protocol that must be formalized at high level of detail (each single byte).
- we have to interact with a wireless NIC that transmits and receives packets on a channel that can be carrying high traffic, so we must be able to filter out messages at MAC layer.
- to inject layer II frames we need to communicate directly with interface driver, since different wireless interfaces require different drivers there must be an easy way to integrate the code with new hardware driver.
- a common situation encountered to perform certain tests is the need to use two wireless devices, one for sending frames and one another for reading them, therefore we must manage different devices and synchronize them to create a correct sequence of events.

While there are various publications focused on protocol testing to verify standard conformance (see [3], [4] for an overview) the only security targeted work we found is the Protos project, described in section III-B.

III. S.T.R.E.S.S.

S.T.R.E.S.S. is an acronym meaning *Stress Testing and Reverse Engineering for System Security*, it's a software for automatic generation of test suites that respects the requisites needed for performing tests on networking software, as illustrated in the previous section.

S.T.R.E.S.S. is targeted to the generation and injection of tests for layer II applications but it can be used at any level

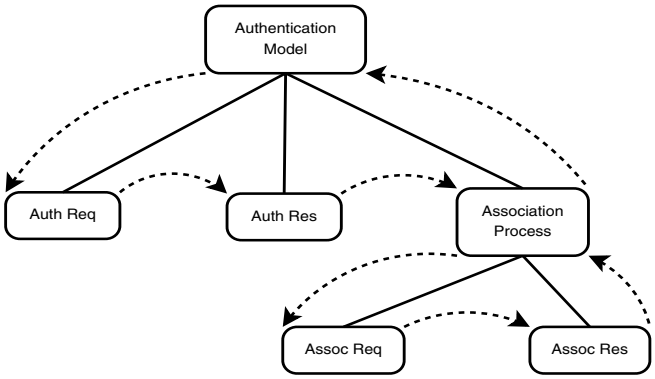

Fig. 2. ABNF tree for 802.11 authentication and association

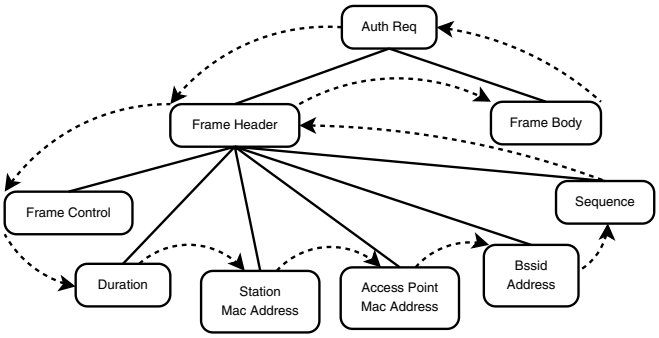

Fig. 3. ABNF tree for authentication frame, this tree expands the node labeled Auth Req in fig. 2

```
AuthenticationModel = AuthReq AuthRes
                      AssociationProcess
AssociationProcess  = AssocReq AssocRes
AuthReq     = FrameHeader FrameBody
FrameHeader = FrameControl Duration
              StationMacAddress
              APMacAddress Bssid Sequence
FrameControl = 0x0080
FrameBody = ...
```

Fig. 4. Example ABNF code

of the OSI stack; before the test a set of anomalies are chosen and the platform itself generates a number of tests sufficient to combine them all.

To obtain maximum generality and re-usability *S.T.R.E.S.S.* is not bound to a particular communication standard but it is able to use any protocol defined with a generic grammar. To represent a protocol we used a meta-syntax, a formal language to describe it. The chosen meta-syntax is the *Augmented Backus-Naur Form* (*ABNF*, see [5]), an IETF standard specifically designed to represent complex Internet protocols that perfectly suites also for layer II protocols, whose structure is often simpler.

With ABNF a communication protocol is described as a tree data structure; as an example in fig. 2 and 3 it is represented authentication and association phase of plain 802.11 protocol. At a high level of abstraction the handshake is made of 4 packets:

- From Client station to access point: Authentication request
- From Access point to Client station: Authentication response
- From Client station to Access Point: Association request
- From Access Point to Client station: Association response

if the tree in fig. 2 is traversed from top to bottom and from left to right (following the arrows) the handshake is reproduced. This simple representation intuitively explains the packet exchange needed by the handshake.

At a higher level of detail, each of the frames will be composed by a message header and eventually a message body, as specified in IEEE 802.11; figure 3 represents the authentication request frame (6 fields in the header, from *frame control* to *sequence* and eventually the frame body). Figure 3 substitutes the *Auth Req* block in fig 2, again if traversed in the correct way it intuitively represents the handshake in a higher detail. The same tree is reported, as ABNF code in figure 4 where recursive definitions describe to a increasing detail the protocol structure.

While internal nodes represent syntax and alternatives, leaf nodes normally represent data to be sent or received by the platform. In ABNF syntax the word beginning the line is interpreted as a command, so that if the definition of a node starts with `send` or `receive` a packet will be sent to or received by the interface. Data from received packets can be saved and used as variables, so that more complex commands such as crypto functions or CRC can be implemented,

Using this language *S.T.R.E.S.S.* can be programmed to perform a conversation with a real access point, not blindly injecting packets but forging correct packets and interpreting received data to perform different actions (taking a certain direction into the tree). This dynamism makes it easy to check whenever the device under test is behaving correctly; since *S.T.R.E.S.S.* interprets the received frames, it can verify that the responses are consistent with the standard and with the internal state machine. For example the machine under test might answer with a frame containing a header value that doesn't correspond to what the standard mandates in that moment of the execution, or might not respond at all (a configurable timeout is started for every expected frame).

So far we explained how *S.T.R.E.S.S.* can be instructed to realize a correct communication with a standard compliant device, for operating real stress tests we must insert inside the communication some anomalies. An anomaly might be a field containing a wrong or unusual value according to the protocol, as an UTF string in a field that should contain an ASCII string, or a field longer than its declared length. Software bugs are normally hidden in parts of the code that are rarely executed and might cause a failure in interpreting the data. We try to find them injecting anomalies in different parts of the communication, so the first step is to individuate some points in the protocol that are suitable to be *exploited*. Once

```
...
eapol2 = "send" %x08013a01 Bssid Sta Ap
         %x909d llchdr wpaversion wpatype
         lenght2 descriptorType keyInfo2
         keyLenght replay1 nonce2 KeyIV
         WPAKeyRSC WPAKeyID WPAKeyMIC2
         WPAKeyLenght WPAKey
; Second frame in WPA 4 way handshake
; for WPA-PSK in standard IEEE 802.11i
WPAKeyLenght   = CorrectValue /
                 AnomaliesValues
CorrectValue    = %x0018
AnomaliesValues = %x0000 / %xffff
...
```

Fig. 5. Example ABNF code with anomalies for WPA-PSK

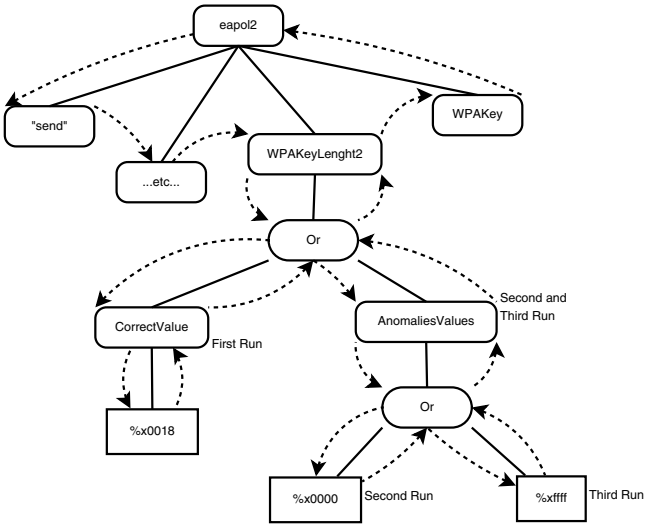

Fig. 6. ABNF tree for WPA-PSK example

we have a list of entry points and for each of them a list of anomalies, all these information are added to the *ABNF* grammar as alternatives to the original, correct grammar.

Therefore the final tree has all the information about protocol syntax, packet structure, point of inclusion and type of anomalies needed to perform the test.

As an example we consider the *4 way handshake* for *WPA-PSK* authentication defined in *IEEE802.11i*. Our aim is to stress test the implementation of various access points emulating the behaviour of a client machine entering the network. Therefore we create a model for a WPA supplicant in *ABNF* and then we select entry points for fault injection.

The *4 way handshake* is composed of EAPoL packets and it is started by the access point, we decided to put anomalies in the frame sent by the client as a response to the first packet. In that frame there are two interesting fields, a *WPAKey* and *WPAKeyLeght*, the first one contains a key to be exchanged between the machines, the second defines the length of the first field, we try to trigger an underflow or overflow error inserting a null value `0x0000` and e a huge value `0xffff` into this last field.

In figure 5 it is reported the ABNF definition of the second EAPoL frame of the *4 way handshake*. The frame is constituted by a number of uninteresting fields and the two fields described before, the *WPAKeyLeght* field is expanded with a *CorrectValue* (hexadecimal 0x0018) or (*or* symbol in ABNF is represented by the slash) two possible anomalies, (hexadecimal 0x0000 and 0xffff). In figure 6 the derived tree is reported, the *or* symbol instructs the platform that the two sub-trees are alternatives, so each of them will be traversed in a different execution. To be completely explored, the handshake will be repeated three times, the first one with the correct value and following ones with the anomalies. If anomalies are added in multiple points as in this example *S.T.R.E.S.S.* will run enough execution to combine them all, if we had added another anomaly with only one incorrect value, our program would generate six test cases.

As said it is possible to call various commands to execute particular actions like frame sending and receiving. In the previous example of *ABNF* model, we used only the *"send"* command; a more complex function is used to authenticate data using a message integrity function as required by the standard with the following syntax:

```
WPAKeyMIC2 = "hmac_md5" kck dataMic2 %d16
```

It is out of the scope of this paper to enter in the details of the implementation, for more information about functions and grammar definition we suggest to visit the SVN repository of the project [6]. However, we want to focus on three key features described so far:

- The platform is flexible enough to perform a complex handshake with a device. As seen, cryptographic functions can be easily added, so that any authentication protocol implementation can be tested.
- It is fundamental to note that whenever an *ABNF* file is realized , the resulting test suites can be used to test *any* number of compatible devices. We tested the *4 way handshake*, with access points of different brands, so that the platform offers maximum re-usability.
- The chosen approach guarantees that once understood how to describe a protocol with ABNF, any protocol representable with that syntax can be tested.

A. Software Planning

S.T.R.E.S.S. is completely written in C++ and released with a GPL license, the platform is split in two separate parts:

- ABNF interpreter
- Interface with multiple devices

S.T.R.E.S.S. takes as input the ABNF specifications file, it validates that file and then recreates in memory the logical tree. To every node of the tree is associated a specific C++ method that will be called whenever the node is traversed, this way we can easily separate single functions, such as syntax elements or commands. This high modular structure is easily expandable, adding new syntax constructs or functions. Special

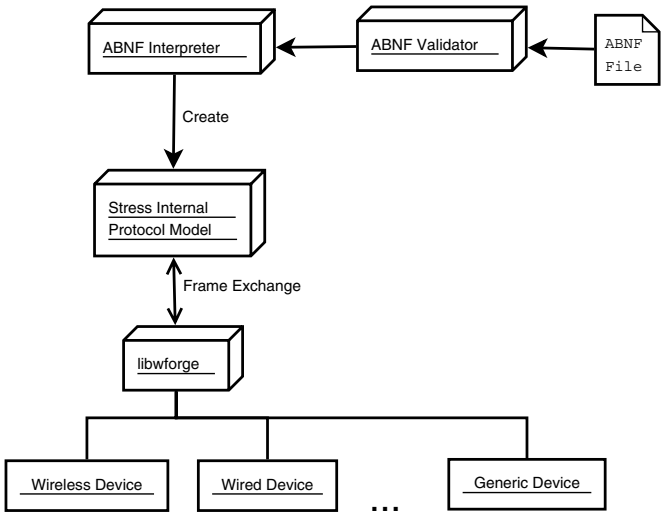

Fig. 7. Software organization for *S.T.R.E.S.S.*

attention has been devoted to re-usability of the code, using Design Patterns [7] for code development and planning.

Interaction with physical interfaces is realized with a specific C library for forging wireless frames called libwforge, developed together with *S.T.R.E.S.S.*. Libwforge is able to manage the interface, pre-filter unwanted packets based on rules that *S.T.R.E.S.S.* defines, sending and reading frames.

S.T.R.E.S.S. structure is represented in fig. 7, it has been designed to be extremely modular so that it is easy to add new interfaces. A new interface might be a wireless device that requires a different driver (so far driver for PrismII and Atheros chipset have been developed) or some completely different hardware NIC, such as a bluetooth or wired ethernet card. It might also be some higher layer socket, such as a TCP connection; due the generic protocol description used once realized the interface any TCP based protocol might be tested. Another important feature we implemented is the possibility to filter packets based on dynamical interaction between *S.T.R.E.S.S.* and libwforge. The platform can instruct the library to receive packets according to a certain pattern to be matched against selected fields in the frame, with this feature we greatly reduce the possibility to evaluate in the stress test a frame which is not part of the communication under test.

B. Protos

In present literature, a similar approach to the same problem can be found in the Protos project [8]. The project has produced a software to perform stress tests for high layer protocol implementation in network software. It is a java and tcl based platform that produces and injects a large number of test cases with an automatic process.

Protos project has been used to test several network protocols with positive results, in most cases it was able to detect some critical vulnerabilities in commercial devices (see [8] for test results and bug publications). Protos uses BNF syntax, that is a simpler version of ABNF with less representation capabilities; once defined the grammar and the entry points for the anomalies it produces all the possible test-cases and saves them off-line, afterwards, it injects the sequences one by one. Packets are blindly injected, so there is no on-line evaluation of received packets, after the completion of all the test-cases the analyst must review all the logs produced by protos and eventually by the device under test to verify the results.

As said, protos software has achieved important results in isolating bugs in various software and hardware, *S.T.R.E.S.S.* is inspired to Protos with the aim of expanding its possibilities and making it more complete and usable. The main difference can be summarized as:

- Layer II usage: Protos is based on java sockets, so it doesn't approach wireless layer II testing. *S.T.R.E.S.S.* has been planned and used for interaction with wireless devices and can be easily expanded to any protocol and device.
- Run-time evaluation: blind packet injection can be sufficient to test protocols with low complexity (especially request/response protocols), on the other hand *S.T.R.E.S.S.* is able to interpret data received from the device under test allowing us to communicate with complex protocols that need dynamic interaction, such as authentication protocols. Moreover off-line production of the tests produces a high amount of data, while *S.T.R.E.S.S.* dynamically produces tests so doesn't need high storage or memory requirements.
- Log evaluation: *S.T.R.E.S.S.* produces logs in a human readable form and is able to detect failures in the responses of the tested device.

IV. Results

During our experiments we used *S.T.R.E.S.S.* to test some software and hardware products; in this section we report the most significative results of the tests done with Cisco Aironet 1200 access point and with the hostapd [9] software access point. For both platforms the authentication phase has been tested, being the most delicate and complex part. Three kinds of authentication have been tested, standard shared-key authentication, authentication with WPA shared key (the so called four-way handshake) and the first packets (identity request/response packets) of the EAP-TLS packets. This last test has been performed to verify the interaction between the two access points and the RADIUS server connected to them (FreeRADIUS software).

A. Cisco Aironet 1200

The aim of this test was to verify the standard compliance of this product with regards of the RADIUS protocol. In particular in a custom EAP modification developed in our laboratory (see [2]) we were adding some information to the identity field of the first EAP packets. In our modifications we include in the identity field a security *TOKEN* with the following syntax:

Length	Identity	EOL	*TOKEN*

where:

- lenght: is the total lenght of the frame fields
- Identity: is the EAP required field
- EOL: end of line symbol (0x00)
- *TOKEN*: security token we need for our custom protocol

If correctly implemented, we expect an EAP/RADIUS compatible access point to evaluate the lenght field, pack all the data into a RADIUS packet and forward it to the authentication server. The authentication server itself will read the identity field until the EOL symbol, then if aware of our modifications will read the *TOKEN*, otherwise it will just drop these informations. We successfully used the *TOKEN* with hostapd based access points without any modifications to the hostapd software, but we could not use it with Cisco hardware. Using *S.T.R.E.S.S.*, with multiple test we were able to reconstruct the behaviour of this device, performing a successful reverse engineering. We realized that Cisco hardware does not consider the lenght field but interpretates the identity until the EOL character, drops the following data, reformats a new EAP packet with modified lenght and encapsulates it into a RADIUS packet. This behaviour makes it impossible to use our modifications with cisco hardware even if protocol compliant.

Other less relevant results showed that the device didn't consider the value of certain fields of the 4-way handshake, such as the Descriptor Type or Key Info.

B. Hostapd software

Hostapd is a software to emulate the behaviour of an access point when using devices that might not support access point features. As an example, wireless NIC's based on Prism II chipset include a *Master* mode in which the NIC behaves as a minimally functioning access point. With hostapd the NIC can be enriched with more functionalities. Hostapd supports several different hardware and it is used in many products based on an embedded GNU/Linux operative system. In the stress tests performed we feeded the software with sequences of malformed packet, testing shared-key authentication and WPA 4-way handshake. Test results showed that hostapd software seems to be more respectful of the protocol compared to products by Cisco but also less stable. In a couple of occasions the device was completely blocked, but we were not able to understand who was responsible for the block, if hostapd, the driver or the physical device itself.

The most serious bug we encountered is related to the faulty processing of the *WPA key length* field. We observed that a field lenght with the value 0xFFFF caused hostapd to exit with a segmentation fault error. The error was due to the fact that when copying the *WPA key* field into system memory, hostapd didn't check if the field lenght corresponded to what *WPA key length* field declared. In this way declaring a lenght longer then the actual lenght of the *WPA key* field hostapd was writing in a buffer shorter then needed. This kind of bug can be exploited to produce a denial of service attack but it might also cause more serious damages, considering that unchecked buffer are the primary cause for remote exploits. After our experiments we filed a bug report to the Debian security report team [10] that has been resolved with a security patch.

V. Conclusions and possible developments

Software bugs are the major cause of security incidents, virus and worm propagation and consequent crimes on Internet. In the latest years, not only the great diffusion of such dangerous software has been encouraged by an underground market based on the trade of software insecurities, but also an independent, professional activity of bug-hunting is considered commercially promising. As a consequence, software houses and open source communities spend constantly increasing energies in finding and patching vulnerabilities. In this paper we presented *S.T.R.E.S.S.*, a platform aimed at finding bugs in software implementations of network protocols, planned to achieve re-usability, modularity and dynamic analysis of results also for Layer II network testing. We observed that testing network software through blind packet injection may be enough for simple request/reply protocols, but modern layer II protocols need active interaction to be able to make a complete conversation and test all the hidden features of the protocol. *S.T.R.E.S.S.* is able to interpret a grammar describing any protocol to be tested and dynamically answer to complex requests. This is fundamental to explore any possibility the protocol offers and to have an on-line interpretation of the results. Once realized a test suite, the test can be performed on any number of compliant devices; last *S.T.R.E.S.S.* is fully modular, so it can be expanded to support any network card or protocol.

Our results shown the effectiveness of our approach, revealing important non standard behaviour and severe vulnerabilities in commercial and open source protocols on both the tested platforms.

References

[1] Institute of Electrical and Electronic Engineers, Inc., *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 6: Medium Access Control (MAC) Security Enhancements*, IEEE Std., 2004.

[2] L. Maccari, R. Fantacci, T. Pecorella, and F. Frosali, "Secure, fast handhoff techniques for 802.1x based wireless network," 2006.

[3] Sidhu and Leung, "Formal methods for protocol testing: A detailed study," *IEEETSE: IEEE Transactions on Software Engineering*, vol. 15, 1989.

[4] D. Hogrefe, "Main issues in protocol testing," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, no. 4, pp. 397–400, Aug. 2003.

[5] D. H. Crocker and P. Overell, "Augmented bnf for syntax specifications: Abnf," RFC 2234, 1997.

[6] L. Maccari and M. Rosi. Repository of source code for s.t.r.e.s.s. project. [Online]. Available: http://lart.det.unifi.it/Members/rosi/stress/

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, P. Education, Ed. Addison-Wesley, 1995.

[8] R. Kaksonen, "A functional method for assessing protocol implementation security," Technical Research Centre of Finland, VTT Publications 447. 128 p., Tech. Rep., 2001.

[9] J. Malinen. hostapd: Ieee 802.11 ap, ieee 802.1x/wpa/wpa2/eap/radius authenticator. [Online]. Available: http://hostap.epitest.fi/hostapd

[10] Debian bug reporting system. [Online]. Available: http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=365897