DYNAMIC POLICY ENFORCEMENT FOR SOFTWARE DEFINED RADIO

Patrick Flanigan, (Security Engineer, NCSA, University of Illinois Urbana-Champaign, flans@ncsa.uiuc.edu), Von Welch, (Senior Security Engineer, NCSA, University of Illinois Urbana-Champaign, vwelch@ncsa.uiuc.edu) Meenal Pant, (Software Engineer, NCSA, University of Illinois Urbana-Champaign, mpant@ncsa.uiuc.edu)

ABSTRACT

Our research analyzes security policy enforcement issues inherent to handheld Software Defined Radio (SDR) devices. We have developed an abstraction for Dynamic Policy Enforcement (DPE) for a SDR system which consists of three distinct modules that monitor changes in external conditions, validate system configuration based on those conditions and a given policy, and implement changes to ensure policy compliance. In order to demonstrate the viability of our system, we created a prototype that implements the roles and responsibilities of our abstraction in conjunction with a prototype SDR system previously developed by NCSA that is based on the GNU SDR software.

1. INTRODUCTION AND MOTIVATION

Typically, standard radio devices have been built for extremely specific functions, limited by the use of narrow bandwidths and rigid hardware specifications. Software Defined Radio (SDR) allows for functionality previously statically-cast in hardware to be implemented in software. The power of SDR lies in the ability to dynamically reconfigure its functionality by changing flexible software. With this flexibility, we achieve tremendous advantages over hardware-only platforms because software can be developed to perform the complex tasks and dynamically updated, altered or even removed based on changing conditions, users or policy. However, with these gains in flexibility, security policy enforcement becomes a major concern. Our work is focused on the design and implementation of a Dynamic Policy Enforcement (DPE) for SDR security.

There are a number of distinct security issues with SDR. The focus of each issue is dependent upon a balance between the required flexibility and the level of security that is desired. There has been a fair amount of prior work focused on allowing for secure dynamic download and installation of software into a SDR and the protection of the base SDR software from malicious code. Our focus is at a higher level of abstraction – the implementation of secure and dynamic policy enforcement for SDR to ensure that the functional pieces of software deployed adhere to policy dependant on the user of the SDR and the conditions in which the SDR is being used.

SDR policy enforcement must take into account dynamically changing users, conditions, environments and needs. In order to decide if a given configuration, by which we mean combination of software in use and parameters such as broadcast frequency, protocol and power, there are a number of factors that effect how the SDR should behave:

- **Who is holding the SDR?** What is the role of the holder of an SDR device? Is it, for example, an average citizen, a responder, a member of law enforcement, or the commander of the response?
- **What are the environmental conditions?** Is it a normal day or is there a condition alert or is there an emergency response going on in the immediate area?
- **What policies are in effect?** Policies would seem to be more static than the previous factors, but may vary in time or as the device moves from one region to another, changing administrative jurisdiction.

It is key to notice that, in particular with the first two criteria, these may change dynamically and outside the control of the SDR device. This requires policy enforcement to not only consider requested changes (e.g. in broadcast parameters or software installation), but factors that change outside the device's control (e.g. who is holding the device or the state of emergency).

Consider the scenario of first responders from an emergency agency (e.g. fire, medical, law enforcement) using SDR-based handheld radios during a response. When the emergency is declared, the first effect might be that average citizens holding SDR-based cell phones or two-way radios would be severely limited in how they could use those devices in order to preserve bandwidth for responders. The responder's devices however, should remain fully functional, or even increase in functionality, allowing them to access normally private channels to facilitate cross-agency communication or to allow for (or even require) encrypted private communication.

And, if under some unusual circumstance, a responder were to use a device belonging to an ordinary citizen, that device should readjust, granting as much access to that responder as permitted by its policy and its implementation.

In this scenario, it is clear that we have many possibilities for authorization and configuration. There could be any number of users with different levels of access, and these users could change at any time as the radio is passed from person-to-person. A change in the alert condition may require a change in basic functionality, for example, reducing functionality to preserve spectrum space for critical services. Different modules for encryption may need to be changed on the fly for higher security. And software modules themselves may have specific policy restrictions as well. The intersection of all of these factors can become quite complex. Because of this, a dynamic and reliable policy enforcement solution is required. By creating an abstract security layer that interacts with the SDR application layer, we can isolate and manage these security and configuration needs. This needs to be accomplished at a layer that is independent of the SDR so that we do not compromise the implementation of the radio device.

Our goal was to design and implement an enforcement system that is capable not only of vetting changes prior to their occurrence, e.g. a user requesting a change in frequency or the application of an encryption scheme, but after their occurrence as well, e.g. the radio is dropped and picked up by a different user. This requires not only traditional policy-based authorization gateways that vet requests, but active system monitoring which validates all elements of the system (user, software state, external environment, etc.) against policy and is capable of making changes to ensure policy is enforced in the face of changes outside the control of the system. First, in Section 2, we present our architecture and design of a Dynamic Policy Enforcement for SDR. In Section 3 we describe our implemented prototype to validate our design. In Section 4, we cover the NCSA implementation of SDR. We conclude with a discussion of related work.

2. POLICY ENFORCEMENT ARCHITECTURE

Our Dynamic Policy Enforcement (DPE) system is focused upon preventing intentional or unintentional behavior on the part of the SDR user, which violates policy in regards to the methods of use of the SDR system. We contend that an independent management system composed of three abstract roles can successfully accomplish this. We call these roles the *monitor*, the *implementor* and the *validator*. These roles are implemented as separate modules that intercommunicate.

The monitor is the entry point to the entire system. It detects and handles all changes to or requests for changing the current configuration. This can be done passively by capturing events, either through software or hardware, such as a sensor, or by actively monitoring the activity within the application layer. The monitor passes all events to the implementor.

The role of the implementor is to enact changes to the SDR system, either by servicing requests that are deemed to be valid under the current policy, or in reaction to changes in external environment that have caused the current system configuration to become invalid based on current policy. The implementor communicates with the validator to determine what configurations are permissible under current policy.

The validator contains policy which describes all permissible configurations of the SDR system based on environmental conditions and user attributes. It receives queries from the implementor and responds with the resultant configuration that should be implemented. This configuration may be one requested or may be modified if the a requested configuration is not permissible.

Dynamic configurations can be represented by permutations of the application's components. Some of these components are external because they exist outside the software. A specific person, role or group denotes the current user. The weather is denoted by some well-defined, finite set of possible weather states. The condition is something that is enforced externally to our system entirely, such as an alert status. As a first level of abstraction, we can consider these factors much like the modules of the SDR stack. Let us call the entire set of components and their dependencies our application layer. And it is crucial that this layer is closed. That is, all possibilities are accounted for at any given time. And each of the components is verifiable, so that we are always sure that the component is what it says it is. These components and their configurations can be mapped-out by sets of permutations. We can represent such images of the application layer in a standard, ubiquitous format such as XML. The monitor, implementor and validator can use these images to communicate about configuration decisions. And due to the abstraction of these security policy issues, we are able to concentrate and isolate authorization and module replacement apart from the application itself.

3. POLICY ENFORCEMENT SYSTEM

We now turn to the detailed design of our system. Our work is composed of two distinct systems – the Dynamic Policy Enforcement (DPE) modules and the SDR modules. How these two systems interact is the focus of our research. This section provides an overview of the DPE implementation, while the next is about the NCSA implementation of SDR. Our intention was to build a prototype of the two systems coexisting on a handheld

SDR. For our purposes, it has been sufficient to build them both on a Linux box (Fedora 4) using GnuRadio 2.5. First we will look at the individual components, their roles and how they intercommunicate.

Instead of using sensor mechanisms to provide external requests, we created a web-based interface that allows a user to select a controlled set of parameters. This GUI was built with Python and consists of three dropdown menus that allow the user to select an 'external' request. We decided to use the role of the user, the current alert condition and the weather as typical parameters for this study. Many more variables could have been used, some perhaps more relevant for specific reasons, but our intention was to keep it simple. We wanted to demonstrate functionality – not complexity. We use specific permutations of these parameters as criteria for the security policy decisions. When changes are selected, they are reflected by rendered diagrams showing the resultant configurations of the SDR. We will also follow a typical flow of a request as it is processed by the DPE system.

3.1 Individual Roles and Responsibilities

In Section 2, we discussed a viable abstraction for a dynamic policy enforcement system which could manage and make security policy decisions for SDR. We built modules which implement this abstraction. The module names were shortened for brevity. The names are montor (monitor), imptor (implementor) and valtor (validator). They are C++ modules that use named pipes to communicate with each other using a messaging API developed specifically for this project.

Montor is the entry point and the event sink for the DPE system for external requests and events fired off by the SDR. Its job is to handle requests for configuration changes and to monitor the SDR so that configuration changes can be made in case there is some failure in the current SDR setup. The motivation behind the event monitoring is so that the SDR can be watched to ensure stability and security. We foresee that threats upon a system can be detected as internal configuration changes or requests and they can be inspected as such. This 'monitoring' capability is not within the current implementation of montor, but it is certainly an area that invites further research.

In the Linux workstation version of this system, the GUI provides external requests to montor. The GUI also allows the user to start and stop the DPE system and the SDR. Each is started as a whole using multiple forked processes. This was implemented for demonstrative purposes. In an actual handheld SDR, these requests could be enabled with sensors designed to detect changes that should be handled by DPE. An example is the use of biometrics to determine a user change that may require a change to the SDR configuration.

Imptor is the workhorse for the DPE system. It handles requests that have been picked up by montor, then gets them validated by communicating with valtor, then actually makes the actual SDR configuration changes as needed. It is aware of configuration needs through the use of specific XML-based configuration files. These files contain resource information used to set up the configuration such as module names as well as repositories for module downloads. They are also capable of ensuring that the correct/secure modules are being used for a specific configuration.

The final piece of the Dynamic Policy Enforcement system is valtor. Valtor assumes the role of the validator. It receives requests from imptor, opens and inspects the XML-based policy file, then processes the request with a configuration that is appropriate for the given parameters and sends it back to imptor. The security policy file can be updated 'on the fly' as well. The current policy that is being used is based upon eXtensible Access Control Markup Language (XACML). We will discuss this further in our Related Works section. We wanted to rely upon a standardized way of representing our security policy that could be applied to SDR.

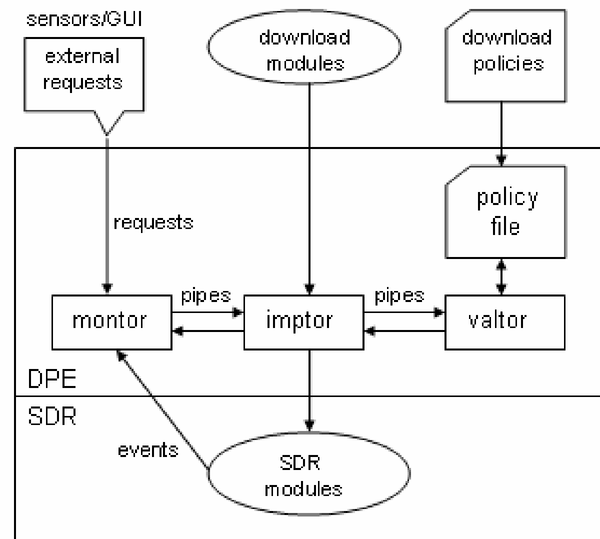


Figure 1: An Overview the Entire System

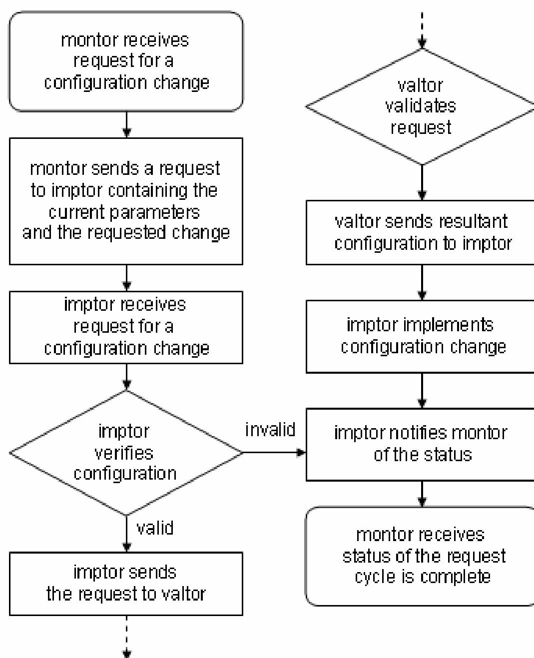
3.2 The Request Data Flow

The primary vision that guided the development of this system is that each module has its own unique role and responsibility. They do not need to know anything about what the other modules are doing. They watch their

message pipes and react, process and reply. In fact, one can shut a module down and the entire DPE system will stop. But if the module is then restarted, the system starts moving again. The entire process is driven by the messaging that is passed through the named pipes. Each module processes its message by reading a pipe, and then completes its role by writing out to another pipe. Figure 2 is a diagram of the flow of a request through the DPE system. The message API between the modules is quite simple. The data is written and read via pipes using a message buffer layer which is mapped into structures which pass the appropriate parameters.

Let's take a quick walk through the diagram. Montor is waiting for something to happen. It receives a configuration change request. In our case, this is very simple because we only have two configurations that are possible. We are only swapping out encryption/decryption modules which reflect the security level given the current parameters – the user, the alert condition and the weather. Montor takes this state information and relays it to imptor. Imptor checks the appropriate configuration file and verifies that it is a valid configuration. If not, imptor replies to montor that it was an invalid configuration request. Nothing is changed. However, if successful, the request gets passed to valtor.

Figure 2: The Request Flow



Valtor receives the request and opens the XML-based security policy file. It checks the permutation of the requested parameters and determines if it is valid for the requested configuration. Valid or invalid, valtor sends a resultant configuration back to imptor. If valid, the requested change is returned. If the policy criteria is invalid, the 'best alternative' configuration is returned. This ensures that even if the security level is not what is required for the request, an appropriate configuration is returned. Thus, the SDR always remains operational. The policy file must be built so that any request will return a valid configuration. It certainly can be flagged as a failed request, but continuous operation without user intervention is a positive alternative to SDR shutdown.

Consider this scenario: the handheld SDR is being used by someone who has a high security clearance. His credentials match the current encryption scheme that is implemented by the SDR configuration. He loses the radio and it is found by someone who has inadequate credentials for the configuration. Montor picked up the change of user, but it has no knowledge of the current configuration. By the time the request gets to valtor, the configuration is passed as the current one, which fails for the new user. A less secure configuration is passed back to imptor and the change is made. The user does not even need to know about the change. It can all be mapped out in the security policy file. This ensures that all final decisions about accessibility are made according to the policy file.

4. NCSA SDR IMPLEMENTATION

In order to demonstrate the viability of our Dynamic Policy Enforcement upon SDR, we have implemented it in conjunction with a SDR system. For our implementation, we are using GnuRadio 2.5 as the underlying software required for implementing a SDR. In order to demonstrate a typical SDR application, we previously developed a reconfigurable software radio 'data stack' that consists of four executables that are inter-connected via UNIX pipes. [3] The data stack is essentially a collection of modules/layers such as source, sink, software defined radio (SDR), *encryptor* and *decryptor* that inter-communicate through a well-defined API. (see Figure 3).

The purpose of this module setup is to provide a data stack that is dynamically reconfigurable. That is, any layer can be replaced at runtime. We can replace modules for security needs such as encryption or replacing modules dependent upon functional needs that vary according to the weather, conditions or the user. And when these modules are swapped, allowable configurations and proper authorization will need to be considered. This requires a standard way of representing all possible configurations of the modules and any other determining factors. We refer to all modules and factors as 'components' of the application layer. These components comprise the SDR.

The impetus behind building such a system is to keep the design fairly simple and well-bounded. Encryption and decryption are somewhat ubiquitous and easy to swap and analyze for a given system. Modifying the SDR behavior in other ways such as bandwidth usage or output power is much more complicated and raises even more security considerations such has regional regulations and standards. Thus, the SDR model for our purposes is intentionally simple and straightforward. We needed a simple model so that we could focus on dynamic policy enforcement without sidestepping into other security issues. If the SDR is built with modules, then our system can be used with it. The layout of the SDR can be specified in the configuration file.

As the name suggests, Source is a data provider, such as a file source or an audio source. Data from a transmitter's Source goes to the intended receiver's Sink, such as a file sink, audio sink etc. The Encryptor module, located at the transmitter, is responsible for data encryption before the data travels through the air interface. This module is implemented as a software object providing a specific encryption scheme, such as Triple DES, AES etc. The Decryptor module, located at the receiver, will decrypt the data before sending it down to the sink. Similar to the Encryptor, this module is also implemented as a software object providing a specific decryption scheme. The SDR module provides a transmit/receive path, filtering and modulation schemes for the data to travel through the air interface.

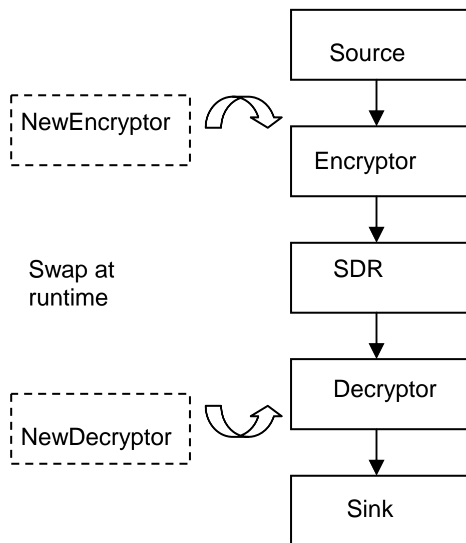


Figure 3: The reconfigurable data stack

4.1. Previous NCSA work with SDR

The reconfigurable data stack was first implemented at NCSA using GnuRadio 0.9 [2], C++ and UNIX pipes. This stack is comprised of Application, Session, Security, Radio Manager and Radio Hardware layers. It is reconfigurable at runtime as well. The modules communicate using named pipes. For a detailed description of the architecture and implementation of this stack please refer to [3].

4.2. Current Implementation

For the current implementation, GnuRadio 2.5, Python 2.3.4 [4] and UNIX sockets [5] are used. Our previous work could not be reused, as the GnuRadio 2.5 code base went through a major implementation change from GnuRadio 0.9. Each of the modules are Python objects. The SDR module is an extension of the GnuRadio 2.5 code. The modulation scheme used by transmit and receive paths is Frequency Shift Keying (FSK). The SDR module receives encrypted data from the Encryptor. This data is then filtered, interpolated and modulated. The signal is then transmitted over the air interface. At the receiver the signal is filtered and demodulated to extract the original data. The modules talk to each other via UNIX sockets. During runtime any of these modules can be swapped out with another similar module seamlessly, based on the command received from the Policy Enforcement front end.

5. RELATED WORK

In this section we briefly review and contrast some related work in the field of SDR security.

The Next Generation (XG) program [6] is developing specifications and concepts related towards using SDR technology for a dynamic redistributable spectrum. Their proposed architecture [7] bears strong similarity to GNU Software Radio-based data stack design, which is not surprising since it also seems to be inspired by the ISO network stack model. The XG group also has a proposed policy language [8] (for which they have a prototype [9]) with implied policy enforcement architecture. This architecture is fairly similar to ours, with a "Policy Conformance Reasoner" corresponding to our validator, an "Accredited Kernel" playing the role of implementor, and the notion of a "Sensor" which partly fills our monitor role. The major differences between the projects are that the XG has spent considerable effort in developing what appears to be a comprehensive policy language and our project has incorporated external environmental conditions besides radio spectrum use, e.g. alert level and device user.

Lam et. al. [10] propose a "Radio Security Module" for validation and lifecycle management of software on a SDR. This work is complementary to the work described in our paper as it serves to validate downloaded software,

while our work strives to manage that software's use under different conditions once it is installed on the SDR.

Hill et. al. [1] have performed a threat analysis on the GNU Software Radio [2] which forms the basis of our prototype implementation. Their work focused on a number of software vulnerabilities within the GNU implementation. These problems include memory access threats and the risks associated with the manipulation of the execution graph. This analysis pointed out some execution weaknesses of the GNU implementation which effect our implementation as well, namely the single address space in which the different software modules run, which in our implementation includes the policy enforcement modules as well. Advancement of our work beyond the prototype phase would need to address this concern. Their work also stressed the need for a strong policy driven configuration that would provide a framework to minimize the risks associated with the programmability of RF parameters. It is crucial that operating constraints be in place so that security policies can be effectively enforced.

6. CONCLUSION

We have presented a possible architecture for dynamic policy enforcement for a SDR system which takes into account dynamic attributes external to the SDR device such as the device user and environmental conditions such as level of alert. Our architecture consists of three main components, which serve to monitor the current system configuration and accept requests for changes to that configuration, validate configuration changes, either requested or externally driven, and then implement changes based on requests or deviation of the configuration from what is valid under policy. To demonstrate the validity of our system, we have prototyped our architecture in conjunction with a GNU Software Radio-based application data stacked previously implemented at NCSA.

We examined at length the problems and constraints that were encountered in this development. As an extension to our findings, we also considered the viewpoint that attacks upon an application\system and internal failure could be seen as changes in behavior that can be detected by the monitor. These could certainly be interpreted as requests for new configurations that could be handled just as safely and easily as we have shown above. Transitioning our focus into software and away from hardware dependency has brought along many inherent security issues. This is seen very clearly with SDR. Our research has been focused upon abstracting these issues out of the application layer and addressing them independently. We have found that secure software systems can be represented in a model that is highly adaptive and configurable. Our prototype provides us with a strong, dynamic security policy enforcement solution for SDR.

7. ACKNOWLEDGEMENTS

This work is funded by the Office of Naval Research through the National Center for Advanced Secure Systems (NCASSR), as was the previous work developing the GNU SDR Radio-based data stack described in [3]. The GNU SDR Radio software base provided the foundation for our project.

8. REFERENCES

[1] R. Hill, S. Myagmar, R. Campbell, **Threat Analysis of GNU Software Radio,** World Wireless Congress (WWC) , May 2005.

[2] http://www.gnu.org/software/gnuradio/

[3] A. Betts, M. Hall, V. Kindratenko, M. Pant, D. Pointer, V. Welch, and P. Zawada, **The GNU Software Radio Transceiver Platform**, Procs of 2004 Software Defined Radio Technical Conference (SDR Forum), Phoenix (AZ), Nov 2004, Vol. C, pp. 41-46.

[4] http://www.python.org/

[5] W.R.Stevens, "Unix Network Programming, Volume 1: Networking APIs - Sockets and XTI", 1997, Prentice Hall PTR

[6] "XG Overview", visited September 29th, 2005, http://www.darpa.mil/ato/programs/xg/overview.html

[7] XG Working Group, "The XG Architectural Framework, "Request for Comments Version 1.0" http://www.darpa.mil/ato/programs/xg/rfc_af.pdf

[8] XG Working Group, "XG Policy Language Framework Request for Comments Version 1.0", http://www.ir.bbn.com/projects/xmac/rfc/rfc-policylang-1.0.pdf

[9] "The BBN XG Projects", visited September 29, 2005. http://www.ir.bbn.com/projects/xmac/pollang.html

[10] Chih Fung Lam, Kei Sakaguchi, Jun-ichi Takada, and Kiyomichi Araki, "Radio Security Module that Enables Global Roaming of SDR Terminal while Complying with Local Radio Regulation," 2003 Fall IEEE Vehicular Technology Conference (VTC 2003 Fall), Oct. 2003 (Orlando, FL, USA).