A Secure Programming Policy for Alma Common Software
Technical Report

Mauricio Araya
Universidad Técnica Federico Santa María
Av.España 1680
Valparaíso, Chile
maray@inf.utfsm.cl

30 September 2004

Abstract

Security Policies are rules that protect a system from malicious intentions. A Secure Programming Policy is presented in this paper as a first draft for a consistent security policy for ACS.

1 Introduction

This paper is intended to serve as a preliminary analysis for several security issues in the Alma Common Software, specially those regarding Secure Programming.

The intended audience are mainly ACS developers and users; nevertheless any large open source project like ACS should be interested in the topics discussed here.

The Atacama Large Millimeter Array (ALMA) is an astronomical joint project of several organizations over the globe. It will consist of a 64 12-meter antennas array working as one at the Atacama desert. To fulfill the software requirements of such a complex control system like ALMA, ACS emerges as a collection of common patterns in control systems and components, providing a middleware (framework) for developing the ALMA software.

In the ALMA project, system safety and security are very important issues: all systems will be constructed using reliable and supported hardware, networks will be protected by firewalls and critical information will be encrypted for transmissions.

But in security the weakest link can break the chain: ACS does not have a well documented policy about secure programming, making the software exploitable and insecure. The main idea of this paper is to detect those possible
exploitable sections, discuss why they are dangerous, and develop a very simple policy draft to avoid them from the beginning.

ACS is an open source software, distributed under the GNU LGPL licence, so most of the common confidentiality policies are not applicable. In general terms, all "Security through Obscurity" policies are vanished from OSS (Open Source Software). A solution to that problem is to use a Secure Programming Policy to avoid classical exploitable code.

2 Previous Work

This paper merges two research areas of Systems Security with a wide set of documentation, papers and books: Secure Programming and Security Policies.

The first area describes guidelines and recommendations on how to write secure code, including topics like input validation, buffer overflow avoidance, code auditing, calling external resources, security APIs and frameworks, data formatting, random numbers and character encoding. Books like [?], [2] and [3] are three different approaches on how to write secure code. For instance [?] covers several techniques and strategies for avoiding programming errors, protecting applications from reverse engineering and using cryptography properly. In [2] the authors focuses they work on how to maintain the security on each stage of the entire code lifecycle, applying security principles through architecture, design, implementation, testing and operation stages. [4] on the other hand, explains how to use the platform features that Java provides to enhance security – the class loader, the bytecode verifier, the security manager, digital signatures, and access controller – using the given APIs like JAAS and JSSE.

Despite, the most applicable work to this paper about Secure Programming is [1], because in addition to the topics described in [?] it includes a new perspective to the security problem: the open source development. An open source project obeys to other forces than the classical full-commercial software. For instance no mistake can be hidden from a potential attacker, so secrecy driven development is not an option. The development in most of the cases is distributed, so many programming cultures and techniques are used in a single project. But the bazaar model ([?]) is not an obstacle to security: in general terms the Open Source software improves the security just due to the freedom of applying patches to the code and to the pressure that involves that the code will be available to anyone.

The second area is Security Policies, including topics like confidentiality, integrity, availability, accountability, risk management, and formal security models. Most of the investigation and development in this area comes from different organizations that strongly need a heavy security model because of their internal structure and classified information that needs to be managed; this includes governments, the army or big companies. For instance [1] was written for military purposes and nowadays it is a fundamental bibliography to understand discretionary access control policy.
The most famous commercial standard is iso 17799 based on BS7799-1 that presents a detailed set of rules and objectives regarding System Access Control, Computer & Operations Management, Physical and Environmental Security, Business Continuity Management, etc. But this standard focuses on certifying a company, so the rules and objectives are thought on how to protect organization entities and objectives; no much information about how to develop secure programs.

This paper will use an informal policy style – no Policy Definition Language will be used – but if a similar policy should be implemented for real, a PDL must be chosen. For more information about PDLs and Secure Policies in general, please refer to

3 Secure Programming Policy

Security Policies usually describe a set of patterns that a trusted system should follow to become secure. A system could be a single computer, a network or even the whole company, and the patterns will depend on what and how much we want to protect. Secure Programming focuses the analysis on programs, describing several patterns and recommendations to achieve a secure program. Then, a SPP (Secure Programming Policy) should include the following elements:

- **The System.** The scope of a SPP are programs, so the system in this case will be reduced to a program, a module, a class or just a function.

- **The Targets.** Classic Security Policies distinguish between objects, subjects, access permissions, messages, etc. SPP in the other hand, deals with low level targets like input data, instructions, functions and buffers.

- **The Patterns.** Patterns in SPP are for program components (targets), so suitable patterns will be those that protect our program from malicious agents.

3.1 The System

A program, a module or a class are very complex structures to apply a direct SPP, but all of these are composed by functions/methods. A function has all the components of the systemic view: input, output, subsystems, external calls, objects and messages. In figure [1] the following elements are shown.

- **Input :** The parameters that the function receives. In general terms, all input should be considered as untrusted.

- **Instructions :** The operations of a function. It can be defined by the programming language, can be a system call given by the Operating System (or library), or it can be another function.
Figure 1: A systemic view of a Function.

- **Calls**: Some instructions make external Calls. This calls can be trusted or untrusted. Calls are always attached to an instruction.

- **Variables**: Variables are used by instructions to fulfill the operations. Variables can be trusted or untrusted depending on what they represent.

- **Resources**: Variables can represent local or external resources. Local resources are trusted. External resources can be trusted or not depending on the policy sensibility.

- **Output**: The output must be trusted. The idea is that other functions select our function as a trusted call.

The two principles in which relays a secure function are: *make a trusted output from an untrusted input* and *protect the instructions from untrusted calls and resources*. The most paranoid model considers all resources and calls as untrusted, but this model is utopian: no programmer will follow this model because in a real function almost all instructions use external calls or external resources. The first task then is to detect a set of calls and resources that are trusted and mark all the others as untrusted. This must not be done at the inverse way. The next stage is to detect which variables are affected by untrusted resources and mark them. With this information the programmer can decide if the instruction that he is writing is clear or tainted. Table 1 shows how to decide if an instruction is clear or tainted. In the second case a *security block* must be written to sanitize the operation.

### 3.2 The Targets

A policy should include subjects and objects. Subjects are unpredictable entities, objects are the "things" that subjects uses. In SPP, a subject will
be an instruction and an object will be a variable. Variables and Instructions can produce a security risk in our function, but variables are easy to test and fix, so in general terms a SPP should protect variables from malicious calls, in other words SPP protects objects from subjects. Then, our main targets will be variables.

### 3.3 The Patterns

The patterns are guidelines to protect our targets from malicious intentions. This paper will introduce five example patterns based on the ones described on [1].

1. **Divide tainted instructions**: An instruction can use more than one call or variable. The idea is to write the same instruction as a set of instructions to write a specific *security block* to the single subinstruction that taints the composed instruction. A *security block* can be variable-based or instruction-based, and will depend on which problem is intended to fix. Doing this, the number of targets affected are minimized.

2. **Validate all input parameters**: Parameters are hosted in variables, so the targets are affected by definition. Sometimes, a *security block* would be not enough to protect us from an attacker, but if the language supports strong type definition, we can set bounded parameters. In any case (bounded parameters or security block) the idea is to set up limits based on the legal values. Again, it is a bad idea to identify illegal values and denied them, it is much better to restrict the parameter only to the legal values.

3. **String parsing**: A string that is well limited (bounded parameter for instance) can still be dangerous when it is interacting with an instruction. Do not consider a string trusted before checking that the char encoding is acceptable, and special characters (like wildcards) are well parsed.

4. **File Descriptors**: In *Unix*, resources are represented by files, so file descriptors are variables that represent trusted or untrusted resources. If a program opens a file is almost natural to check if an error happens, but if the file descriptor was previously opened (like the standard input/output descriptors) the data channel must be checked to protect our targets.
5. Buffer Overflow (C/C++) : Fixed length buffers can be very dangerous. One instruction interacting with an untrusted input, can write outside the buffer's boundaries and execute malicious code through our program. The solution in this case is obvious: trusted instructions (functions) must be used, that means functions that check the boundaries of the operation.

4 Testing ACS Programs Security

ACS Arquitecture is divided in 4 layers, and it is based on CORBA middleware and modules that interacts with the ACS core. First layer are external tools that ACS uses as a base for developing the rest of the software. All these tools will not be included in the analysis (compilers, configuration controls tools, debuggers, CORBA Middleware, ACE, TAO, etc), because they are not directly developed by the ALMA team. The Second layer corresponds to the Core Components that structure all the system abstraction to the upper layers. If a security hole exists at this layer, all the ACS system could be compromised, because this layer includes the main ACS Component abstraction that all further components will depend on. The third layer are services that interacts directly with untrusted input and programs, like the Command System, Management Access Control, etc. The fourth and last layer provides high level APIs and tools for developing the components that will manage the hardware and applications. For more information about ACS Architecture, please refer to

The source code is divided in software modules (CMM/CVS) and each module is part of a layer (see figure ??). The idea of this section is to identify which modules (and layers) are affected by typical security problems as ones described in ??, using automatic auditing tools. There are two free tools that are widely used in the open source community, FlawFinder and RATS. The first was written and maintain by David Wheeler, same author of ??, so the results are based on the topics discussed on that text. The problem is that FlawFinder only supports the ANSI C language, making the study far incomplete.

4.1 RATS - Rough Auditing Tool for Security

RATS was developed by Secure Software, supporting much more languages than FlawFinder, but (as its name says) ”RATS performs only a rough analysis of source code. It will not find all errors and may also flag false positives”. Several modules were tested with RATS, but we consider, for this study, only the higher security alerts that RATS provides. The full output can be found in the appendix. The result is disturbing: almost all core components are compromised with classical programming mistakes. This result does not mean that every module is exploitable, but the risk of an attack is imminent.
Table 2: RATS: Fixed Size Local Buffer.

<table>
<thead>
<tr>
<th>Module</th>
<th>Layer</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>acscomponent</td>
<td>core</td>
<td>2</td>
</tr>
<tr>
<td>acserr</td>
<td>core</td>
<td>14</td>
</tr>
<tr>
<td>acslog</td>
<td>core</td>
<td>6</td>
</tr>
<tr>
<td>acstime</td>
<td>core</td>
<td>4</td>
</tr>
<tr>
<td>acsutil</td>
<td>services</td>
<td>3</td>
</tr>
<tr>
<td>baci</td>
<td>core</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 3: RATS: Malformed String Operation.

<table>
<thead>
<tr>
<th>Module</th>
<th>Layer</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>acserr</td>
<td>core</td>
<td>8</td>
</tr>
<tr>
<td>acslog</td>
<td>core</td>
<td>3</td>
</tr>
<tr>
<td>acstime</td>
<td>core</td>
<td>5</td>
</tr>
<tr>
<td>acsutil</td>
<td>services</td>
<td>9</td>
</tr>
<tr>
<td>baci</td>
<td>core</td>
<td>3</td>
</tr>
</tbody>
</table>

4.1.1 Fixed Size Local Buffer.

Buffer Overflow is the most common technique to gain access to execute malicious code. A fixed size local buffer without the right validation can be very dangerous. Table 2 resumes the RATS output about FSLB. "Extra care should be taken to ensure that character arrays that are allocated on the stack are used safely. They are prime targets for buffer overflow attacks". Core components like acscomponent or baci are affected by this problem and almost all further components will inherit theses classes: if the method is exploitable, then all the software is exposed to the same issue.

4.1.2 Malformed String Operations.

If an instruction that uses strings is not well used, the attacker could add formatting characters and get confidential information. Table 3 shows the components (and the occurrence) of this problem.

This type of problem includes C functions like `printf`, `strcpy`, `strcat`, `sscanf`, `fscanf` and `printf`. 
Table 4: RATS: Other Problems

<table>
<thead>
<tr>
<th>Module</th>
<th>Layer</th>
<th>Problem</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>acspy</td>
<td>core</td>
<td>eval</td>
<td>17</td>
</tr>
<tr>
<td>acssim</td>
<td>services</td>
<td>eval</td>
<td>4</td>
</tr>
<tr>
<td>acspy</td>
<td>core</td>
<td>execfile</td>
<td>1</td>
</tr>
<tr>
<td>acspy</td>
<td>core</td>
<td>gethostbyname</td>
<td>2</td>
</tr>
<tr>
<td>acsutil</td>
<td>services</td>
<td>getenv</td>
<td>6</td>
</tr>
</tbody>
</table>

4.1.3 Others

Other important problems were detected regarding the relation of the functions with the outer world. Table 4 shows the components (and the occurrence) of each problem. Environment variables are very dangerous (getenv), and must not be used without a strict security block to protect the rest of the code. Other problems are eval and execfile (Python), because untrusted arguments can exploit those "simple" instructions to run any code. At last, gethostbyname can be easily deceived by an attacker, so the programmer should not trust in DNS for critical communication.

5 Applying a SSP to ACS

Using the idea described in section III, the following Policy (written as a cookbook) is presented.

To write a Secure Function/Method:

1. Define the scope of the method and minimize its functionality.

2. Select a group of trusted resources and calls, and mark all others as untrusted. The methods and variables of the object must be considered as trusted. Be very careful with inheritance, a secure method in an untrusted object is worthless.

3. Define the function with bounded parameters.

4. Write the next instruction and check if is tainted with \[?\] .

5. Instructions (or functions) that do not provide length validation must be considered as tainted.

6. Instructions (or functions) that use dynamic output formats must be considered as tainted.

7. Functions that call environment variables, or make a direct access to the shell system, must be considered as tainted.

\[1\text{http://www.nl.debian.org/security/}\]
8. Then, if the instruction is tainted, a security block must be written. Use the patterns described in the previous section.

9. If the function is not finished, then go to step 4.

10. At last, run your code through any well written analysis software (like FlawFinder or RATS), consider the error reports and fix every single issue.

6 Future Work

This paper introduces SPP, but it does not give a formal description. A complete SPP definition, with a formal DPL is needed to make a more serious analysis. RATS does not support Java, reason why no problems where reported from the interface layer. A special (Commercial) Java code auditing tool is needed to complete this study.

Closing up, it is obvious that the problems detected here, must be addressed by the ACS developers.

7 Acknowledgments

This work was supported by project ALMA-CONICYT 31060008, and could not been done without the support of ALMA, ESO, NRAO and UTFSM.

References


