Security Code Review for Address Space PKI Repository Project

mudge@bbn.com
July 31, 2007

Introduction / Overview

From the beginning of the work embarked upon for the BBN Address Space PKI REPOSITORY it was determined that a security code review was to be included as one of the project steps. The rationale for this is straightforward and simple but all too often overlooked, even in security relevant programs.

Frequently the end user of an application or service is in the situation of needing or wanting security to such an extent that they will require new tools and solutions to provide this security for them. Such is the case for ensuring the validity of ownership for autonomous system numbers and the address ranges they are permitted to advertise. However, if the user is not able to perform a security analysis of the source code for this new tool, or if the solution is provided without source, then the user is left trusting that the tool or solution: a) solves the problem they are attempting to address, and b) does not introduce new security problems with the introduction of new code. Sadly, case "b" is often incorrect[1][2][3].

If the program is being released with access to the source code, code sections will commonly be lifted and reused in other applications. Thus, it behooves the developers to consider secure coding guidelines in their work so as to minimize the propagation of errors that might exist in the initial code base to new projects.

It is for these reasons that BBN decided to include some form of security source code review of the Address Space PKI Repository program. What follows is a description of what this analysis included and, just as importantly, what this analysis did not address.

Security Analysis

As with any subcomponent within a larger project or activity, one must determine what the correct level of effort for the subcomponent should be in order to maximize the value to the overall project without adversely affecting the project at large. Security audits are no exception.

---

[1] Kerberos4 – L0pht Security Advisory Nov 22, 1996. http:// http://attrition.org/security/advisory/l0pht/l0pht.96-11-22.kerberos
[2] DataLynx suGuard – L0pht Security Advisory Jan 3, 1999. http://attrition.org/security/advisory/l0pht/l0pht.99-01-03.suguard

[3] "For the fifth time in two months, security researchers have publicized a serious flaw in a widely used virus-scanning program.". http://news.zdnet.co.uk/security/0,1000000189,39191831,00.htm

As this project is in many ways a prototype and initial reference base, a full security source code review would be cost prohibitive in both time and finances. Conversely, not performing a security analysis would be remiss. If the code is to be referenced by others then it needs to show a security conscientious approach to coding to encourage others to engage in similar practices and to increase trust in the underlying code.

From these constraints the appropriate level of effort has been determined to be an automated lexical security audit followed by human analysis of the key areas identified.

Automated Lexical Analysis

For the automated analysis, tools previously authored by Mudge were used. These tools take C source and header files as input and parse the contents looking for a number of trigger code segments. These code segments are either commonly misused function system function calls, common library calls that have security risks associated with them, or text that has been discovered empirically to be involved with or within close proximity to security coding errors of varying types. Over ninety (90) functions are identified as being security risk areas within the automated analysis tool and around twenty (20) tags endemic to security issues are also included.

The goal of the automated analysis tools is not to determine which functions and routines are secure and which ones are not. Rather the goal is to identify routines that are commonly insecure and routines that are designed to be secure but frequently used in insecure fashions. By identifying these areas that have a higher probability of containing security vulnerabilities, the amount of code that a human needs to manually inspect is greatly reduced (in comparison to the total code base of the project). As the automated tools are designed to err on the side of identifying a source file and code fragment and displaying it even if there might be other mechanisms in place to protect the code from being exploited, it is hoped that the tools provide too much information rather than omitting important data.

Automated analysis was performed on 35,790 lines found within ninety-six (96) files of the Address Space PKI Repository project. From this 2,619 lines of code were highlighted as being potential security concerns.

Human Analysis

With the identified lines of code from the parsed files, the next step involved a human looking not only at the lines identified, but also tracing input and output variables that interact with or are created by the suspect code fragments and examining the code surrounding these suspect sections.

The human is then able to determine which code segments are actually exploitable and which are not. It should be noted that even if the code segment is not exploitable, good practices would encourage that the code still be modified to make it blatantly obvious that

this is not the case. Such activities show the reader of the code that the authors are performing security conscious coding practices and eases future security code auditing activities.

While this effort does not embody full code coverage (and further it does not guarantee that all security problems have been brought to light), it does a responsible job of identifying common errors. The result is an identification of a substantial amount of vulnerabilities that would require either low or moderate sophistication to exploit. This was the effort deemed appropriate given the project environment and constraints.

Highlights and Summary

Unbounded copies

The vast majority of errors identified were unbounded string copies. These were often operating on addresses located on the program stack or heap. These included standard vulnerable unbounded calls such as strcpy, sprintf, strcat, and others of similar nature. Some of these were exploitable whereas the majority were not. It was recommended that each of these be replaced with the bounded version of the function in question (e.g. strncpy, snprintf, strncat, etc.), even if the code in question was not obviously exploitable.

Bounded copies

In a few locations bounded string copies were found to be suspect and potentially vulnerable due to incorrect termination tests and improper length specifications. In the case of incorrect termination tests the majority were found to be within loops utilizing pointer arithmetic under the assumption that a marker value would be contained within the source data. In the event that the marker value was not found there was no subsequent test to ensure that the destination memory address would not be overrun. For situations where improper length specifications were found these took the form of inadvertent specification of either the size of the source data to be used as the maximum value to place in the destination data location, or a specification of external sizes (often in #defines) where the size was greater than the allocated memory in the destination referenced for storage. The latter being a somewhat common coding area where a function such as strncpy(dest, src, len) has the len value specified as the sizeof(src) rather than the sizeof(dst). These situations were pointed out to the appropriate authors for remediation.

Known Problematic Functions

A small number of errors were identified where function calls that previously had been considered safe but later found not to be so were used. Examples of such calls are mktemp(3) and stat(2). In the case of mktemp(3), the filename that will be returned by the call can be guessed. As the filename produced is only guaranteed not to exist at the point of creation of the string, a race condition exists that can oftentimes lead to varying degrees of system exploitation depending upon the program purpose and context. Good

security practice encourages the use of mkstemp(3) in place of mktemp(3). For stat(2) a similar race condition exists in that the values returned from the call are only valid at the exact time that the call is made and may not be correct when an action is later taken on the file in question. If possible, it is recommended security practice to use fstat(2) in place of stat(2).

Format String Attacks

There was one case where input that could be affected by external sources, and that was trusted to specify format characteristics, was detected. The result is that unfiltered user input is provided as the format string parameter for certain functions. The attacker can utilize format tokens such as %x to print data from program memory and tokens such as %n to write data to program memory. The code identified could result in a format string attack and was brought to the attention of the developer for remediation.

Permissions and File Creation Errors

A limited number of locations where unnecessarily lax permissions were used in the creation of files were identified and pointed out to the developers. Similarly, there were a small number of places within the code where the third argument to open(2) was omitted. In this latter case of file creation, open would use values located on the program stack as input for the file creation mode with the result being arbitrary permission values.

A common error in the creation of files was also found. If a file is created via the open(2) call with the flag of O_CREAT specified, a new file will be created. In many cases where file creation is desired, the flag O_EXCL should be used in addition to O_CREAT. This additional flag ensures that the file that is about to be created does not already exist and that the last component in the pathname provided to open(2) does not reference a symbolic link (even if that link points to a non-existent file). The absence of this flag would enable to the attacker to cause the program to create files with names and paths specified by the attacker. If the program is running with elevated privileges this can cause system compromise and denial of system wide services.

Shell Execution

The most interesting error found through these processes was that of task handoff between the running program and the underlying system. In particular that of handing off tasks to be executed by system command interpreters.

In the case identified, part of the data comprising the command being passed to the external interpreter was retrieved from remote sources and thus should have been treated as potentially tainted. If a malicious person were to encode this data with various shell meta-characters it would have been possible to remotely compromise the Address Space PKI Repository system. As the account running the program is expected to have complete access to the database, at the very least total database compromise could occur along with user level compromise of the host system. If the account running the program was

operating with elevated privileges total remote system compromise could have been attained.

The developers were alerted to this and provided several suggestions for solutions. The first was to introduce several hundred new lines of code from the open source application rsync(1). This was the program and argument string that was being passed to the command interpreter. This would be the safest solution but would also require the most amount of re-work and effort. The next solution was to introduce string parsing functions that would be used to sanitize the arguments being passed to the command interpreter. This would not require as much effort as the first suggested solution with the caveat that it would not provide as much potential protection and such efforts have historically been shown to be difficult to implement correctly. The final suggestion was to document this security issue in a highly visible fashion so as to dissuade people from recreating such situations elsewhere and to demonstrate that this is an understood issue that is being tolerated for the time being due to the prototype and proof of concept nature of the program.[4]

Level of Effort and Completeness

This analysis attempted to approach the task of identifying problem areas within the code base in a fashion similar to that performed by many attackers. Out of thousands of lines of code within the total project, the sections most likely to contain vulnerabilities were identified. From this point, the majority of effort could be spent addressing these areas rather than focusing on sections that were unlikely to contain vulnerabilities. This approach attempted to address the most commonly found coding errors that could be leveraged to compromise the integrity of the program and of the system the program operates within.

This effort does not guarantee that all security vulnerabilities were identified. Nor does this effort provide full code coverage and discovery of complex logic vulnerabilities.

Modifications and Corrective Measures Taken

The majority of issues identified were corrected using the suggestions provided during the security analysis. These included such practices as replacing unbounded string copies with bounded versions, ensuring the most restrictive yet usable permissions are applied to file creation, protecting against race conditions, and other similar solutions. There were however a few areas where corrections were not made. The reasons behind these being a combination of time and effort required given the available funding and the low potential risk.

For example, the *casn* library received modifications due to its potential for ubiquitous use. Preventing malicious acts in this code base was deemed important. *asn_gen*, on the

[4] In this case, the second option was chosen with comments introduced to the source code to make other developers aware of the potential security caveat.

other hand, is a tool that is only used when ASN.1 definitions change. This is further only used to generate “C” code in such situations. Given the difficulty of implementing some of the suggested bounds checking within this code, and the fact that said code base is not part of the running application and cannot be used for exploitative purposes the modification of *asn_gen* code was omitted.

As the system is specific in purpose, it is believed that real world use would primarily be performed on dedicated systems. Thus certain race conditions inherent in common system calls such as stat() were attempted to be minimized. As these are low risk vulnerabilities, replacing stat() with fstat() and subsequently passing and maintaining file descriptors between functions was viewed as too large an effort for small gain.