SANS Institute
InfoSec Reading Room

Secure Software Development and Code Analysis Tools

The first half of this document discusses secure coding techniques. The main languages chosen to facilitate the discussion are Perl, Java, and C/C++. These were chosen due to their popularity and extended usage in the software development community. The latter section of this document contains the results of the research and tests conducted on some freely available source code analysis tools. All these tools have a common objective: To quickly scan source code for potential security issues and to communicate them to th...

GIAC Certification:
Practical Assignment v1.4b
Option 1

Secure Software Development and Code Analysis Tools

Author: Thien La
Date: September 30th, 2002

SUMMARY **4**

CODE REVIEWS **5**

SECURE PROGRAMMING GUIDELINES **5**

GENERAL GUIDELINES 5
Input Validation *5*
SQL Statements *6*
Commented Code *6*
Error Messages *6*
URL Contents *7*
Setuid Programs *7*
Strip Binary Files *7*
PERL 7
Taint Checking *8*
The Safe Module *8*
The Warnings (-w) Switch *9*
Setting the PATH Variable *9*
JAVA 9
Printing Messages to Standard Out *9*
Encapsulation *10*
Policy Files *10*
C/C++ 11
Buffer Overflows *11*
Format String Attacks *11*
Executing External Programs *12*
Race Conditions *12*
Checking for Valid Return Codes *13*

SOURCE CODE ANALYSIS TOOLS **13**

PSCAN 14
Conclusion on PScan *15*
FLAWFINDER 15
Conclusion on Flawfinder *16*
RATS (ROUGH AUDITING TOOL FOR SECURITY) 16
Conclusion on RATS *17*
SPLINT (SECURE PROGRAMMING LINT) 17
Conclusion on Splint *18*
ESC/JAVA (EXTENDED STATIC CHECKING FOR JAVA) 18
Conclusion on ESC/Java *19*
MOPS (MODELCHECKING PROGRAMS FOR SECURITY PROPERTIES) 19
The hello.c Example *20*
Conclusion on MOPS *21*

CONCLUSION **21**

APPENDIX A – SOURCE CODE AND SCAN RESULTS **23**

APPENDIX A – SOURCE CODE AND SCAN RESULTS 23

FIGURE 1: TEST.C 23
FIGURE 2: PSCAN RESULTS FOR TEST.C 27
FIGURE 3: FLAWFINDER RESULTS FOR TEST.C 27
FIGURE 4: RATS RESULTS FOR TEST.C 32
FIGURE 5: SPLINT RESULTS FOR TEST.C 34
FIGURE 6: ESC/JAVA RESULTS FOR TELNET.JAVA 38
FIGURE 7: HELLO.C 48
FIGURE 8: HELLO.TRA 48

RESOURCES 50

Summary

The first half of this document discusses secure coding techniques. The main languages chosen to facilitate the discussion are Perl, Java, and C/C++. These were chosen due to their popularity and extended usage in the software development community. This document does not give an elaborate overview of what makes a secure application. That is, it is assumed that the reader has an understanding of the general concepts of authentication, authorization, input validation, logging, error handling, and other application security concepts, and why they are important to the overall security of an application. These concepts instead are intrinsic to the ideas presented herein.

The latter section of this document contains the results of the research and tests conducted on some freely available source code analysis tools. All these tools have a common objective: To quickly scan source code for potential security issues and to communicate them to the user in a detailed, well formatted, easy to understand report. The goal of these tools is not to replace manual reviews, but to facilitate the review process of catching common errors that could lead to security problems. Flawfinder was found to be the most useful tool in terms of the depth and breadth of its scan results, and ease of use. RATS was found to be the tool of choice for flexibility as it is able to scan not only C code, but also Perl, PHP, and Python. Also, the reports that it produced were found to be the most detailed, and easy to understand. Both of these tools offer a good first step towards conducting a manual code audit. Some of the common vulnerabilities they will find are buffer overflows, format string attacks, race conditions, and insecure system calls. For those who want a tool that enforces even tighter checking, try MOPS. MOPS is different because it exhaustively searches through programs line by line to find a path that can cause a security violation (referred to as a violation of a *Temporal Safety Property*). The caveat is that MOPS takes more work to set up, as it requires the user to describe violations via finite state machines.

Code Reviews

Developing robust, enterprise level applications, is a difficult task, and making them completely secure is an impossible task. Fortunately, in reality, security is not about creating an "impenetrable fortress". It is about managing risk (i.e. Let us build a moat around the fortress, have only one way in and out, and post guards on the perimeter day and night).

Arguably, one of the best ways to manage risk for any application is to review its code, and review it *again and again*. Many times, an application is exploitable as a direct result of "lazy" programming, and an indifference to quick "spot checks" and peer reviews. It does not take much to prove this; one only needs to go to the *CERT Coordination Center* web site at http://www.cert.org and type in any well-known, industry recognized application in the search field (try *Internet Explorer* or *Netscape Navigator*). Such searches may result in phrases such as "buffer overflow", "unauthorized access", "execute arbitrary code", "cross-site scripting", and "format string attack". To the inexperienced programmer whose concerns never involved security, these terms may sound far too complex to even consider when designing and coding an application. Function first, security *later*. Yet to the experienced programmer, these types of security issues are all easily avoidable, if only time was taken to understand them.

Secure Programming Guidelines

There are numerous guidelines and tips that a programmer can implement to aid in the prevention of common security bugs in applications. Many of these can be applied to any programming language, but some are specific to only one. The specific guidelines in this document will focus on the Perl, Java, and C/C++ languages. Most of the time, common mistakes can be avoided by a bit of diligence and a good understanding of the underlying issues. Such preventable mistakes often lead to security vulnerabilities that can threaten the confidentiality, integrity, and availability of the information.

General Guidelines

Input Validation

In a client-server environment, perform *server-side* input validation as opposed to *client-side* validation (e.g. Javascript based validation). By placing a proxy between the client and server, client-side validation can be easily circumvented. The proxy would allow the attacker to alter data after it has been "validated" by the client (similar to a "man-in-the-middle" attack).

In terms of implementation, input validation should be done by first defining a set of valid (permissible) characters, and then checking each character of the input against the valid set. If a character from the input is not found in the valid set, the application should return an error page and state that invalid characters were found in the input. The reason for implementing validation in this manner is because it is more difficult to define a set of *invalid* characters and most likely, some characters that should be invalid are not accounted for.

In addition, bounds checking (e.g. maximum length of a string input) should be implemented on top of the valid character check. Bounds analysis helps to avoid most buffer overflow vulnerabilities.

It is worth mentioning that input coming from environment variables should not be trusted and must be validated as well. Also, avoid placing sensitive information (such as passwords) in environment variables. Certain Unix flavors (e.g. FreeBSD) contained a 'ps' utility that allowed users to view the environment variables of any current process, which could potentially reveal confidential information.

SQL Statements

If the application makes calls to a backend database server, make use of stored procedures rather than constructing SQL statements within the code. Embedded SQL statements are especially dangerous if input from outside of the program is used to construct the statement. It is difficult to prevent an attacker from using an input field or a configuration file (loaded by the application) to perform an SQL injection attack. Of course, input validation would help to mitigate such a risk as well.

Commented Code

All commented code should be removed before an application is rolled out into a production environment. Commented code is code that does not belong in the final application as it was either for debugging or for testing. Either way, it should be removed to avoid accidental usage in the production environment (it's not probable that a comment identifier is removed to activate dormant code, but it's possible, and so this is strongly recommended).

Error Messages

All error messages to the user should not divulge any sensitive information about the system, network, or the application. Whenever possible, it is best to use generic messages with error codes that could only be understood by the developers and/or the support groups. An example of generic error message would be "An error has occurred (Code 1234). Please contact your helpdesk."

URL Contents

If it is a web application, never divulge information on the URL such as passwords, server names, IP addresses, or file system paths that reveal the directory structure of the web server. Such information can assist in an attack. For example, this would be an example of an unsafe URL:

```
http://www.xyzcompany.com/index.cgi?username=USER&password=
PASSWORD&file=/home/USER/expenses.txt
```

Setuid Programs

Avoid using setuid programs. Unix systems allow programs to temporarily escalate privileges of a user. Binary files have a protection mask that contains a special bit called the setuid bit. When the setuid bit is *not* set, the executed binary file will spawn a process that runs with an *effective* user ID (UID) of the user executing the file (the user's *real* UID). However, let us assume that the super-user (root) makes a copy of the binary and sets the setuid bit. Now anyone who executes the file will create a process that runs as the *owner* of the file (in this case, root). As a result, the process will access to the entire file system. Consequently, much caution must be exercised when writing setuid programs.

Try not to *setuid root* whenever possible. Instead create a new user ID and use that UID instead (since Unix reserves the first two hundred IDs for internal use, there are plenty of UIDs to be defined). If the UID must change, it is recommended to call *seteuid* (sets only the effective UID of a process) rather than setuid (sets the real, effective, and saved UIDs of a process), since this allows for more granular control of which UID is actually changed. In addition, the effective UID can be reverted back by using the saved UID (which was not changed). Also consider using setreuid (sets only the real and effective UIDs of a processs) as an alternative.

Strip Binary Files

Use the GNU `strip` utility to remove printable symbols from a binary file. This would prevent an attacker from extracting any useful information from the binary file by using the GNU `strings` utility.

Perl

Over the years Perl has easily become one of the most versatile programming languages for both systems administration and Web CGI (although conceivably, you can program *almost* anything using Perl). Its extended usage on the Internet

as a development tool for CGI makes it a popular gateway for attacks on a web server. In addition, the fact that most CGI scripts have escalated privileges compared to those of the user, make it an even more attractive target to attackers. The following list describes some simple proactive and preventive measures a developer (especially a CGI programmer) can easily implement to improve the overall security of Perl code (Please note: This is not a guide to securing a CGI script on a web server. Such a guide can be found in the Resources section at the end of this document).

Taint Checking

Perl version 5.x contains a built in input validation feature called Taint Checking. If enabled, it does not allow user input (any input coming from outside the program) the ability to manipulate other external programs (e.g. piping the data to another program to be executed). Often, a programmer can not trust incoming data (called *tainted data*) that is fed into a script or a program, since there is no guarantee that it will not do harm (intentional or accidental). Taint Checking can be turned on by including a `-T` as a command line switch. For example, you can include `-T` in the first line of any Perl script as:

```
#!usr/bin/perl5 -T
```

Data that would be considered "tainted" includes command line arguments, environment variables, and input from a file. Even variables that reference tainted data become tainted themselves. If the script tries to use tainted data in an insecure manner, a fatal error will be encountered (stating "Insecure dependency" or something to that effect). In some cases, enabling Taint Checking will cause a script to stop running, which is mostly a result of the Perl interpreter demanding that the *full* paths of all external programs referenced by the script be listed in the PATH environment variable and also, that each directory contained in the PATH is not writable by anyone other than the directory's owner and group. Taint Checking's sensitivity to the environment may deter most programmers from using it, but whenever possible, Taint Checking should be used, especially if code is run on another's behalf such as in the case of CGI scripts.

The Safe Module

What if it was not the input that could not be trusted, but the actual code itself? For example, a user downloads an ActiveX control from a web site and it is actually a malicious Trojan horse. Taint Checking would be useless in such cases. The Safe Module allows the programmer to associate Safe objects with different chunks of code in a Perl script. Each Safe object creates a restricted environment for the chunk of code to run within. This is analogous to the concept of using *chroot* that constrains a process to run only in a subdirectory of the overall directory structure. Instead, the notion of Safe objects constrains the

chunk of Perl code to operate only on certain packages in the Perl package structure. Implementation instructions of the Safe module are beyond the scope of this document, but programmers are advised to take advantage of it whenever possible. More information on the Safe module can be found in the Resources section.

The Warnings (-w) Switch

This `-w` enables the display of all warnings when the script is interpreted by Perl. Warnings will be reported on variables used only once or not used at all, undefined file handles, file handles not closed, or an attempt to pass a non-numeric value as numeric. This feature is not specific to security, but could be helpful in debugging errors that could directly or indirectly affect security. In general it's recommended and considered best practice to always use `-w`. You can implement `-w` on the first line with Taint Checking:

```
#!usr/bin/perl5 -Tw
```

Setting the PATH Variable

Set the PATH to a known value as opposed to simply relying on the value at startup. Attackers may use the PATH variable to aid in an attack against the application, like trying to make it execute an arbitrary program. This of course, applies to most other languages as well.

Java

Since its release in 1995, Java has become the programming language of choice for simple to complex web-enabled applications. It was designed with security in mind and as a result, contains features such as a garbage collector for salvaging unused blocks of memory, a strict "sandbox" security model, and a security manager that limits the activities of an application on a particular host. As a result, there may seem to be fewer recommendations for the developer to improve the security of Java code, but the following tips can still make a significant difference.

Printing Messages to Standard Out

For production Internet systems, avoid using `System.out.println()` or `System.err.println()` to print out logging or error messages. The reason to do this is that when messages are printed to standard output, it is difficult to determine exactly where that *is* at any given instant. It is possible to accidentally disclose privileged information to an attacker.

Encapsulation

In Java, if you declare a class, method or field without an access modifier (`private`, `protected`, or `public`), then it defaults to `package` and has access to any class in the same package. One should keep in mind that although this provides encapsulation for the package, this only holds if every piece of code that is loaded into the package is controlled by authorized person(s). A malicious user can insert their own classes and have full access to all classes, methods, and fields in the package.

Java's policy files support two permission prefixes that control access to packages.

- `accessClassInPackage`
- `defineClassInPackage`

By default, all classes in the standard library are accessible publicly (with the exception of classes that begin with "`sun`."). To secure a package, you must edit the `java.security` file in the `${JAVA_HOME}/jre/lib/security` folder. The important line from that file is:

```
package.access=sun.
```

Although this approach works, it contains caveats. For example, the progammer has to be careful when defining the package to be secured in the java.security file. Since the value given to `package.access` is literal, "`sun.`" would protect packages such as "`sun.tools`" but not packages such as "`sun`" or "`sunshine`".

An alternative approach is to use JAR sealing. A JAR (Java ARchive) file is a collection of class files bundled into a compressed format similar to the popular ZIP format. When the class loader loads a class from a sealed JAR file, subsequent classes that are in the same package can only be loaded from that JAR file. To invoke sealing, one must set the seal attribute when creating the JAR:

```
Sealed: true
```

Sealing JAR files is a better method than setting permissions as it does not require the security manager to be installed.

Policy Files

Java's built-in security manager is a convenient tool to use when enforcinig restrictions on applications. Since most of the time, a lot of work is needed to write a custom security manager, JDK 1.2 and later provided a way to *describe* security settings instead of *implementing* them. This was through Java policy

files. With policy files, you can control file system and network access in a relatively granular fashion. For example, you can restrict an application to only have the ability to write to filename `foo`.
It is highly recommended to use Java policy files and the Security Manager instead of trying to "re-invent" a class or system to restrict access to the host and network. The details of how to manipulate policies are omitted from this document, but you can find more information in the Resources section at the end.

C/C++

C is intrinsically an unsafe language. For example, most its standard library string functions are susceptible to buffer overflow and format string attacks if used without caution. In addition, it is a language that is widely used because of its flexibility, speed, and it is relatively easy to learn. The following are some C specific recommendations when trying to develop a secure program.

Buffer Overflows

Avoid using any string functions that do not implicitly perform array boundary checks, as they are susceptible to buffer overflow attacks. The following functions are such functions that should be avoided. Also, each function's respective "safer" alternative is listed as well.

- Instead of `strcpy()`, use `strncpy()`
- Instead of `strcat()`, use `strncat()`
- Instead of `sprintf()`, use `snprintf()`
- Instead of `gets()`, use `fgets()`

In the first three cases the extra 'n' on each alternative function represents the size of the buffer in question. The 'f' on the last function, stands for *formatted* which allows the user to specify the format specifiers for the expected input. These alternative functions force the programmer to define the size of buffer to be manipulated and also the type of input to expect.

Format String Attacks

These kinds of attacks are related to buffer overflow attacks as they often rely on the fact that some functions such as `sprintf()` and `vsprintf()` assume an infinitely long length for the buffer. However, even using `snprintf()` over `sprintf()` does not totally protect a program from a format string attack. Such attacks are carried out by passing format specifiers (`%d`, `%s`, `%n`, etc.) directly in the buffer accepted by the print function. For example, the following is insecure:

```
snprintf(buffer, sizeof(buffer), string)
```

In this case, it is possible to insert format specifiers in the `string` variable to manipulate the memory stack in order to write values of an attacker's choice (such values could contain small programs in themselves to be executed by the processor as a subsequent instruction). More information on how to perform such attacks can be found in the Resources section. The recommendation is to use the following instead:

```
snprintf(buffer, sizeof(buffer), "%s", string)
```

Executing a format string attack is not trivial. Firstly, an attacker must be able to acquire a footprint of the memory stack somehow (by either extracting it from the application or using a debugger), and then the attacker must know exactly how to address specific parts of the memory space in order to manipulate variables on the stack.

Executing External Programs

It is recommended to use the `exec()` function instead of `system()` function to execute an external program. This is because `system()` accepts an arbitrary buffer as the entire command line to execute the program:

```
snprintf(buffer, sizeof(buffer), "emacs %s", filename);
system(buffer);
```

In the above example, the filename variable is exploitable by inserting extra commands to the shell by using semicolons as delimiters (e.g. filename could be `/etc/hosts ; rm *` which would remove all files in the directory in addition to displaying the /etc/hosts file).

The `exec()` function on the other hand, ensures only the first argument is executed:

```
execl("usr/bin/emacs", "usr/bin/emacs", filename, NULL);
```

The above ensures that filename is only fed as an argument into the *Emacs* utility. Also it uses the full path to the Emacs command as opposed to using the PATH variable, which is exploitable by an attacker.

Race Conditions

Generally when a process wants to access a resource (be it a disk, memory, or a file), it takes two steps:

(1) It first checks if the resource is free

(2) If it *is* free, it will access the resource, but if it isn't free, it waits until the resource is no longer in use before attempting to access it.

The main problem arises when a second process wishes to access the same resource some time between steps (1) and (2) above. This could lead to unpredictable results. The processes may lock or this may cause a security hazard whereby one process acquires the escalated privileges of the other process. Attacks are mainly focused on programs that run with elevated privileges (referred to as setuid programs). Attacks on race conditions usually try to benefit from the resources that a program can access while it is executing. Alternatively, programs that don't have elevated privileges are at risk as well, as an attacker may wait for a user with elevated privileges to run that program (such as root) and then attack it.

The following recommendations help to alleviate the problem of race conditions:

- When manipulating files, use functions that use file descriptors as opposed to using functions that use the path to the file (e.g. using `fdopen()` instead of `fopen()`). File descriptors ensure that a malicious user can't use links (symbolic or physical) to change a file while it is open, and before it's actually manipulated by the original process.
- Use the `fcntl()` and `flock()` functions to lock files as they are being written or even read from so that they can not be accessed by another process. It creates virtually atomic operations.
- Use caution when manipulating temporary files. Often it could lead to a race condition. More information on this can be found in the Resources section.

Checking for Valid Return Codes

It is important to check for valid return codes. An example of this is the old implementation of `/bin/login` where the result of not checking for error codes, caused the application to return root access whenever it did not find the `/etc/passwd` file. It was reasonable if the file was damaged, but if the file actually existed and it just couldn't be accessed, then this was a major problem.

Source Code Analysis Tools

When it comes to ensuring that code meets a certain level of security, there are several ways one can approach the problem. As stated previously, one of the best methods of doing this (other than actually conducting white and black box testing in a QA environment) is to conduct code reviews, preferably by as many people as possible. However, sometimes there will be a tight deadline for the

code to be rolled out to production and even the code reviews may miss a seemingly unobvious error. Such subtle errors may lead to significant security bugs in the overall system. For these reasons (and others), it is good to automate part of the review process by using a Source Code Analysis Tool (SCAT). This document presents seven such tools. Each tool was tested using the same test suite (test.c), which is comprised of C and Java code snippets that purposely contain potential security errors. The test results are compared and each tool is carefully scrutinized on the following attributes:

1. *Flexibility* – Tools that have many options and can scan many types of code will score higher in this area.
2. *Accuracy* – The main goal is find security bugs accurately and not have to sort through pages of false positives, or even worst, let false negatives go undiscovered.
3. *Ease of Use* – Most of the time, a programmer just wants to point the tool to the code and click "scan". Programmers don't want to waste time developing complex testing schemes just to do spot checks unless it's absolutely necessary.
4. *Reporting* – The scan results should be displayed in an easy to understand format and preferably with tips on how to fix each of the problems and why they should be fixed to begin with.

The differences in speed at which each tool can parse code is negligible so speed was not considered during the grading (although this does not include the time it takes to *setup* a scan, which in case of MOPS, was a significantly longer than others). In addition, these tools are freely available and in some cases, so is their source code. As a result, cost was never a concern.

PScan

Pscan is a limited scanning tool that will find buffer overflow and format string attack exploits in C code. It will spot all the common `printf()` problems from the Standard C Library such as:

```
sprintf(buffer, variable);
printf(buffer, variable);
```

The explanation of how such statements can be exploited is in the C/C++ guidelines section under Buffer Overflows and Format String Attacks. One limitation of Pscan is that it does not look for traditional buffer overflows which are caused by insufficient bounds verification. However Pscan is fast and accurate with what it sets out to do.

PScan was run against `test.c` (source code in Appendix A, Figure 1) and its results are shown in Appendix A, Figure 2. From Figure 2, you can see that it

catches all the "printf-like" (including `syslog()`) when format specifiers are not explicitly used in the call.

PScan's reporting features are very basic and simple. It simply returns the result to the command line. Of course, you could always pipe this into a file. Also, it does not describe why a particular statement is erroneous; it simply tells you how you may want to correct it.

Some notable features of PScan include the ability to scan multiple files at once. Also, you can specify additional definitions of problem functions by using `-p` on the command line.

Conclusion on PScan

Pscan is simple and easy to use. It finds a very specific kind of error and it finds it fast and accurately. However, PScan would not be recommended for important business applications, because its search scope is too narrow to accommodate a complex application. PScan is recommended more for small snippets of code that are relatively uncomplicated.

Flawfinder

Flawfinder is a static analysis tool that uses a database of known insecure C functions, to scan a program for any potential security issues. It is similar to PScan, but finds many more types of errors. In addition to the `printf()` and standard string manipulation functions, Flawfinder can also find problems with race conditions and system calls. Results are also sorted by risk level for convenience (highest first).

Flawfinder was run against `test.c` (source code in Appendix A, Figure 1) and its results are shown in Appendix A, Figure 3. One will first notice the immense amount of information returned by Flawfinder (127 dangerous functions reported as a opposed to 7 by PScan). Even if half of the reported security bugs were false positives, the amount of information that it returns is still impressive.

Aside from the common buffer overflow errors and race conditions, one will notice that Flawfinder will give warnings for even relatively subtle security problems. For example, it warns that the function `MultiByteToWideChar()` requires a maximum length attribute in amount of characters and not bytes. The warning continues with:

```
Risk is high, it appears that the size is given as bytes,
but the function requires size as characters.
```

The risk here is obviously high, because if the user accidentally gives the size as bytes, the buffer would be much larger than if it was given as the number of characters. A larger than intended buffer size gives opportunity for a malicious user to form a buffer overflow attack.

The amount of detail in the report is very useful for the programmer. Flawfinder even categorizes each vulnerability with either `buffer, format, shell, port`, or `misc` in brackets. At the end of the report, you'll notice that Flawfinder also gives you the amount of time it took, 0.96 seconds for 228 lines of code (although this varies on different hardware, it is still considered very fast).

Conclusion on Flawfinder

Flawfinder is an exceptional source-scanning tool that programmers can depend on to find the most common security problems with C programs. It is fast, and the reporting features are detailed and user-friendly. Installing and using Flawfinder was also relatively simple.

Some notable features are:

- The presence of an internal help menu (use `-help`)
- The ability to tell Flawfinder that certain segments of code should be ignored (use `// Flawfinder: ignore` or `/* Flawfinder: ignore */`)
- Customizable formats for reports
- The ability to give input values as command line arguments to the application
- The ability to save *hitlists* so that future scans of slightly modified code can show the differences between hits before and after modification.

Flawfinder would be recommended as the first of many stages in reviewing simple to complex applications. One complaint about this tool is that it can only scan C code.

RATS (Rough Auditing Tool for Security)

RATS is the only scanner that was tested that had the ability to scan more than one type of language. RATS has the ability to find vulnerabilities in C, C++, Perl, PHP, and Python source code which gives it an edge in flexibility like no other tool that's available (for free). RATS will look for common buffer overflows and race conditions (similar to Flawfinder). RATS also allows reports to be generated in HTML.

RATS was run against `test.c` (source code in Appendix A, Figure 1) and its results are shown in Appendix A, Figure 4. One will notice in the results that

although RATS seems to catch much less security bugs (17) than Flawfinder (127), this does not mean that it is worse than Flawfinder. Instead, it has a different reporting style. For example, it reports on a potential security risk such as `sprintf()` and then displays the line numbers that the same problem is found including a detailed description of the error (this includes what an assailant may use as method of attack). Therefore, it does not report much less, it just reports in a more condensed and easy to read fashion (compared to that of Flawfinder).

Conclusion on RATS

RATS is by far the most versatile of all the tools due to the number of languages that it is able to scan for vulnerabilities. In addition, it also has a nice suite of features which includes (but is not limited to):

- Ability to add XML reporting features using *expat* (see Resources section at the end) XML parsing library (this requires a slight modificiation to the RATS source code)
- Can configure at runtime: level of output, other vulnerability databases (a feature which makes the list of vulnerabilities highly scalable)

One of the limitations with RATS is that its pattern matching algorithm uses a greedy approach. That is, if RATS is scanning for "printf", it will also report on "print" and "vsnprintf". This may at times create a report containing many false positives.

Although, RATS doesn't find as many vulnerabilities as Flawfinder for C code, it definitely makes it up with its ability to scan so many different types of languages, and also by its elegant and detailed reporting features. In summary, RATS is a useful scanner to have in the programmer's toolbox, especially for those who constantly use many different languages.

Splint (Secure Programming Lint)

Splint is a free, "light-weight", static analysis tool that scans for security vulnerabilities in C code. However, unlike the other tools, Splint also looks for coding mistakes and coding style which may not have anything to do with security. Its scanning process is much more involved and complex since it also looks for issues such as undeclared variables, proper return statements, missing arguments, and other stylistic issues (like having comment symbols within comments).

Splint was run against `test.c` (source code in Appendix A, Figure 1) and its results are shown in Appendix A, Figure 4. The first five scans resulted in parse errors (terminating the scan), because Splint complained about many function

calls were not made in any type of construct such as `main()` or another function. Again, this shows the sensitivity of Splint towards style and completeness, which is contrast to simply analyzing single lines at a time regardless of what came before or after them. After changing test.c slightly to accommodate to Splint's sensitivity, it ran and found a total of 82 issues. Most of the issues were not security related and if they were, Splint did not declare it as a "security" issue but rather a warning just like it would declare a style error. For example, Splint warned about the following:

```
test2.c:135:1: No argument corresponding to sprintf format
code 2 (%s): "%s"
```

Although Splint warns the programmer that no type specifier was passed as an argument to `sprintf()` it does not specify that the consequences of not fixing such an error could result in a significant security issue (buffer overflow or format string attack).

Conclusion on Splint

Splint is easy to install, fast, and comes with very detailed documentation (124 pages worth), but from a pure security point of view, it does not deliver (compared to Flawfinder and RATS) in the areas of the breadth of types of vulnerabilities it can find, and the format in which it reports its findings. The report is very general and does not provide any kind of insight as to what is really wrong with the statement (whereas RATS does this very well).

Splint is a great tool for finding errors in all areas of an application, but if a programmer is specifically looking for security bugs, Splint can not compete with the other tools that have been reviewed. As a result, it is recommended that Splint be used along side of another scanner like RATS or Flawfinder, which would re-enforce the search for security vulnerabilities.

ESC/Java (Extended Static Checking for Java)

Developed at the Compaq Systems Research Center, ESC/Java is a static analysis scanner that finds common coding problems with Java applications. ESC/Java scans for errors at compile time and usually finds bugs that are not normally found until runtime (e.g. null dereferences). Also, it employs modular checking techniques that allow it to check code that contains method calls from other classes, even if the code for those classes is not available. Also, it is able to scan libraries with no subclasses defined.

ESC/Java was run against `telnet.java` (the source code was omitted due to its length and its admittance bared no real relevance to the test; the goal was to simply observe the kinds of errors that ESC/Java was capable of finding) and its

results are shown in Appendix A, Figure 7. The test code `telnet.java` is a fully functional telnet client that is contained within an applet. One can observe that most of the errors found were either null dereferences or type cast warnings. Again, there wasn't much in terms of expectations of finding security bugs in Java code as the language itself is very much in contrast with C, assumed to be an unsafe language.

A notable feature of ESC/Java is when it reaches an if-then-else structure, it traces execution through all possible paths in search of errors. This is exhaustive approach is one good feature of ESC/Java.

Conclusion on ESC/Java

Although ESC/Java is a good tool for finding common errors in Java code, it does not focus specifically on security. This may be due to the nature of Java (a "secure" language), and most likely to no fault of the tool itself.

ESC/Java is free and would serve as a good "spot checker" for any Java code.

MOPS (MOdelchecking Programs for Security properties)

MOPS is a free tool that scans C programs for security holes and helps to enforce defensive programming. MOPS is based on the concept of *Temporal Safety Properties* (TSP). A TSP simply describes the order of a sequence of operations. For example, a setuid-root program (a program with an effective UID of root), should first drop its privileges before executing an untrusted program. An object called a Model is used to describe a TSP, and hence the word *modelchecking* in the name MOPS. Modelchecking is a form of static analysis that attempts to scan for violations of TSPs, which in most cases, signal a security bug in the code. The Models are described using Finite State Machines (FSM), and then MOPS uses the FSM to scan the source code for violations. All violations found are printed out as *paths* from the source.

Currently, although MOPS is in its second release, it still remains in its early stages. The current release only comes packaged with the scanner. It is up to the user to develop custom Models using FSMs to describe the TSPs. Future releases are expected to be packaged with a library of common TSPs ready to be used with the scanner.

MOPS was not tested against the test suite (`test.c`), because the test suite is made up of snippets of code that try to cause security exceptions (each line is potentially a security bug), whereas MOPS is more geared towards an entire sequence of operations that amount to an actual application. Fortunately, the MOPS user guide came with example code sequences that were in essence, small programs (they compiled and ran). The user guide also gives instructions

on how to creat FSMs to describe the TSPs used for these examples. MOPS was tested on one of the example applications (Appendix A – Source Code in Figure 7 and Scan Results in Figure 8).

The hello.c Example

The hello.c program (the source can be seen in Appendix A, Figure 8) attempts to get a password entry using the `getpwuid()` function with the real UID as an argument. If it successfully retrieves the `passwd` entry, it proceeds to drop its privileges to that of the real UID. If it cannot get the `passwd` entry, it does not drop its privileges and proceeds to call `execv` to execute a program of the users choice (entered as an argument at the command line). This is obviously a violation of the TSP and should be caught by MOPS.

The steps to have MOPS scan an application are (this is also in the user guide in a slightly different form – see the Resources section):

1. Make sure that the Java `CLASSPATH` environment variable is pointing to the MOPS jar file. First create the environment variable if it does not exist (the `$` symbolizes the shell prompt) and point it to the jar file:

   ```
   $ CLASSPATH=../src/class:../lib/java-getopt-
   1.0.9.jar; export CLASSPATH
   ```

   The export makes sure that processes can take advantage of the change.

2. Parse the source program into a Control Flow Graph (CFG) using the gcc compiler:

   ```
   gcc –B ../rc/ -c hello.c > hello.cfg
   ```

3. Most lines in a program have no relevance to the TSP and so should be removed. They can removed by *compacting* the CFG file:

   ```
   java CfgCompact setuidexec.mfsa hello.cfg
   hello.s.cfg
   ```

4. Now comes the actual “scanning” portion of MOPS or in this case modelchecking. The CFG is checked against the TPS defined in `setuidexec.mfsa` in FSM format:

   ```
   java Check setuidexec.mfsa hello.s.cfg main
   hello.s.tra
   ```

The check should yield “Final states reachable” which means that the final state of the FSM can be achieved and therefore, the TSP Model has been violated.

5. The last step is to transform the path found by the Check into a path mapped out onto the program. This will produce a file named `hello.tra` which contains the violating path (by line number), which can be followed to understand and correct the security bug.

```
java Transform hello.cfg hello.s.tra hello.tra
```

See Appendix A, Figure 2 to view `hello.tra`, which shows the path of the violation. One can follow the path by using the number on each line after the first colon, which is the line number. You’ll notice that MOPS tries all the paths through a program to test for violations. In hello.c, the violation occurs on lines 10 to 14 when the `passwd` entry is not attainable, the function returns and privileges are not dropped.

Conclusion on MOPS

MOPS is not as straight forward to use as the other tools. MOPS requires that you set up tests by yourself, so you’ll need to know how to construct the Model using an FSM (all in the user guide). Also, MOPS requires that you have a Java Runtime Environment installed and working (since the parser is written in C, and the modelchecker written in Java), so the installation takes a bit more work as well.

However, MOPS is definitely a tool all C programmers should have handy. MOPS approaches the problem of finding security vulnerabilities in a unique, yet logical way. Rather than checking for single lines of fault that are caused by `snprintf()` or `syslog()` statements, it takes one step further, and actually checks if there is a path through the program that will enable an attacker to actually exploit the fault. It’s definitely worth the setup time involved to run a scan if an important application needs to be reviewed thoroughly. Its reporting features are also very useful, since it gives you the path (line by line) that would cause the exception. This feature greatly helps the programmer to zero in on the problem and correct it.

Conclusion

All of these tools are good to use as a basic first step towards a formal code review of an application. They allow a programmer to remove much of the common vulnerabilities found in code, but they can never replace a manual review involving a walkthrough of the code with peers and co-workers. One

major problem with static analysis based scanners such as PScan, Flawfinder and RATS is that they do not expand bits of code that contain macros or definitions found in other files, whereas MOPS will actually try to determine the exact paths through the code.

MOPS is definitely a tool to keep an eye out for as it develops further. The next release promises to include all the FSMs that model common security vulnerabilities. With such improvements to the setup time, MOPS has the potential to become a very powerful tool in a programmer's auditing arsenal. In addition the creators of Flawfinder and RATS are planning to merge both tools into a single scanner, which should also prove to be a potent tool for code audits.

Appendix A – Source Code and Scan Results

Figure 1: test.c

This code is courtesy of David Wheeler, developer of Flawfinder, and Alan DeKok, developer of PScan)

```
/*

This is code to be used for testing source code analyzers.
The point of this code is to test the extensiveness and
sensitivity to which the source code analyzers (for C code)
will find security flaws.

The code does not compile which is not important for
these tests.

Mentionables:
Thanks David Wheeler for a lot of this code
Thanks to Alan DeKok (pscan) for his snippets

*/

#include <stdio.h>
#define hello(x) goodbye(x)      //define function hello to act as
goodbye
#define WOKKA "stuff"            //define WOKKA to be a string
containing "stuff"

main() {
     printf("hello\n");
}

/* This is a strcpy test. */
/* Most of these tests are about the fact that these functions
are assumming infinite
   length strings, which is bad if the caller is not careful
about overflowing the buffer */

int demo(char *a, char *b) {
 strcpy(a, "\n");                        // copies "\n" to string a
 strcpy(a, gettext("Hello there"));      // copy string returned
from gettext to a
                                         // Note that gettext is
undefined

 strcpy(b, a);                           // copies contents of
string a to string b
 sprintf(s, "\n");                       // tries to send formatted
output carriage return "\n" to string s
 sprintf(s, "hello");                    // same as above
 sprintf(s, "hello %s", bug);            // note that bug does not
exist
```

```
 sprintf(s, gettext("hello %s"), bug); // depends on what gettext
returns but gettext DNE
 sprintf(s, unknown, bug);             // this is very bad,
allows ..
                                       // attacker to do use
format ..
                                       // specifiers to  mangle
the stack
 printf(bf, x);                        // same as above, but
prints to stdout instead of a buffer
 scanf("%d", &x);                      // reads integer from
stdin
 scanf("%s", s);                       // reads string from stdin
 scanf("%10s", s);                     // same but length is
expected to be 10
 scanf("%s", s);
 gets(f); // Flawfinder: ignore        // get string f (does not
check for buffer overflows)
 printf("\\");
 /* Flawfinder: ignore */
 gets(f);
 gets(f);


}



demo2() {
  char d[20];
  char s[20];
  int n;

  _mbscpy(d,s);                       /* like strcpy, this
doesn't check for buffer overflow */
  memcpy(d,s);
  CopyMemory(d,s);
  lstrcat(d,s);
  strncpy(d,s);
  _tcsncpy(d,s);
  strncat(d,s,10);
  strncat(d,s,sizeof(d)); /* Misuse - this should be flagged as
riskier. */
  _tcsncat(d,s,sizeof(d)); /* Misuse - flag as riskier */
  n = strlen(d);
  /* This is wrong, and should be flagged as risky: */
  MultiByteToWideChar(CP_ACP,0,szName,-
1,wszUserName,sizeof(wszUserName));
  /* This is also wrong, and should be flagged as risky: */
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof
wszUserName);
  /* This is much better: */
  MultiByteToWideChar(CP_ACP,0,szName,-
1,wszUserName,sizeof(wszUserName)/sizeof(wszUserName[0]));
  /* This is much better: */
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof
wszUserName /sizeof(wszUserName[0]));
```

```
  /* This is a terrible idea - the third paramer is NULL, so it
creates a NULL ACL.  Anyway, this needs to be detected: */
  SetSecurityDescriptorDacl(&sd,TRUE,NULL,FALSE);
  /* This one is a bad idea - first param shouldn't be NULL */
  CreateProcess(NULL, "C:\\Program Files\\GoodGuy\\GoodGuy.exe -
x", "");
}

int getopt_example(int argc,char *argv[]) {
    while ((optc = getopt_long (argc, argv, "a",longopts, NULL ))
!= EOF) {
    }
}

int testfile() {
  FILE *f;
  f = fopen("/etc/passwd", "r");
  fclose(f);
}

/**********************from
pscan.c***********************************/

/*
 *  This may be a problem.
 */
fprintf(stderr, variable);    /* problematic */

/*
 *  This MIGHT be a problem, depending on where the 'format'
 *  string comes from, and what it's value is.
 */
fprintf(stderr, format, variable1, variable2);

/*
 *  This is safer.
 */
fprintf(stderr, "%s", variable); /* OK */

/*
 *  Constant strings can't be modified externally, so they're OK.
 */
sprintf(buffer, "string");    /* OK */

sprintf(buffer, "%s");  /* OK */

/*
 *  The variable may contain formatting commands!
 */
sprintf(buffer, variable);    /* problematic */

/*
 *  This is the safe way of doing it.
 */
sprintf(buffer, "%s", variable); /* OK */

/*
```

```
 *  The first sprintf is OK, but the second one has a problem.
 *  This is a check for nested security problems.
 */
sprintf(buffer, "%d", sprintf(buffer1, variable)); /*
problematic! */

/*
 *  strerror(errno) isn't a problem function, and snprintf has
lots
 *  of arguments after the format string, so this is OK.
 */
snprintf(buffer, sizeof(buffer), "test: Error opening %s: %s\n",
filename, strerror(errno)); /* OK */

/*
 *  Multi-line sequences get checked, too.  This one should be
OK.
 */
snprintf(buffer, sizeof(buffer), "test: Error opening %s: %s\n",
       filename,
       strerror(errno)); /* also OK */

/*
 *  This multi-line sequence shouldn't be OK.
 */
sprintf(buffer,
      variable);  /* problematic */

/*
 *  Lots of arguments after the format string.  It's up to your C
 *  compiler to see if you're using the right number of arguments
for
 *  the format string.
 */
sprintf(buffer, "%s %s %s", one, two, three); /* OK */

/*
 *  Nested braces should be OK.
 */
printf((variable ? "%4" : "%3s"), string); /* OK */

/*
 *  User-supplied format strings are OK, I guess...
 */
printf((variable ? fmt1 : fmt2), string3); /* OK */

/*
 *  There's still only one argument for printf, that's a problem.
 */
printf((variable ? string1 : string2));   /* problematic */

// sprintf(buffer, variable); C++ comments get ignored, for good
or for bad.

/* sprintf(buffer, variable); these comments get ignored, too */

/*
```

```
 *  This next bit of code is from the wu-ftp source.  It's OK,
but it
 *  gets flagged because the parser isn't smart enough to check
for
 *  previous, safe, uses of strings.
  */
sprintf(s, "PASV port %i assigned to %s", i, remoteident);
syslog(LOG_DEBUG, s);

/*
 *  The following are references to the functions, but not actual
 *  function calls, so they're OK.
 */
void *foo[] = {snprintf, fprintf}; /* OK */

/*
 *  Your program may define a problem function in one file,
 *  and use a variable of the same name in another file.  We
don't
 *  want to complain about uses of those variables.
 *
 *  I know this won't work in a real C program, but it's a way of
faking
 *  such a variable reference, to ensure that pscan ignores it.
 */
fprintf[1] = 1;               /* OK */

/*
 *  NetBSD allows err(1,NULL).  We should, too.
 */
err(1, NULL);
```

Figure 2: Pscan results for test.c

```
../giac/test.c:75 SECURITY: syslog call should have "%s" as
argument 1
../giac/test.c:134 SECURITY: fprintf call should have "%s" as
argument 1
../giac/test.c:162 SECURITY: sprintf call should have "%s" as
argument 1
../giac/test.c:173 SECURITY: sprintf call should have "%s" as
argument 1
../giac/test.c:191 SECURITY: sprintf call should have "%s" as
argument 1
../giac/test.c:214 SECURITY: printf call should have "%s" as
argument 0
../giac/test.c:227 SECURITY: syslog call should have "%s" as
argument 1
```

Figure 3: FlawFinder results for test.c

```
Flawfinder version 1.21, (C) 2001-2002 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 127
Examining ./test.c
```

```
./test.c:51  [5] (buffer) gets:
  Does not check for buffer overflows. Use fgets() instead.
./test.c:75  [5] (buffer) strncat:
  Easily used incorrectly; doesn't always \0-terminate or check
for
  invalid pointers. Risk is high; the length parameter appears to
be a
  constant, instead of computing the number of characters left.
./test.c:76  [5] (buffer) _tcsncat:
  Easily used incorrectly; doesn't always \0-terminate or check
for
  invalid pointers. Risk is high; the length parameter appears to
be a
  constant, instead of computing the number of characters left.
./test.c:79  [5] (buffer) MultiByteToWideChar:
  Requires maximum length in CHARACTERS, not bytes. Risk is high,
it
  appears that the size is given as bytes, but the function
requires size as
  characters.
./test.c:81  [5] (buffer) MultiByteToWideChar:
  Requires maximum length in CHARACTERS, not bytes. Risk is high,
it
  appears that the size is given as bytes, but the function
requires size as
  characters.
./test.c:92  [5] (misc) SetSecurityDescriptorDacl:
  Never create NULL ACLs; an attacker can set it to Everyone
(Deny All
  Access), which would even forbid administrator access.
./test.c:92  [5] (misc) SetSecurityDescriptorDacl:
  Never create NULL ACLs; an attacker can set it to Everyone
(Deny All
  Access), which would even forbid administrator access.
./test.c:34  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to
destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily
misused).
./test.c:37  [4] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
./test.c:38  [4] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
./test.c:39  [4] (format) sprintf:
  Potential format string problem. Make format string constant.
./test.c:42  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:44  [4] (buffer) scanf:
  The scanf() family's %s operation, without a limit
specification,
  permits buffer overflows. Specify a limit to %s, or use a
different input
  function.
./test.c:46  [4] (buffer) scanf:
  The scanf() family's %s operation, without a limit
specification,
```

```
  permits buffer overflows. Specify a limit to %s, or use a
different input
  function.
./test.c:57  [4] (format) syslog:
  If syslog's format strings can be influenced by an attacker,
they can
  be exploited. Use a constant format string for syslog.
./test.c:68  [4] (buffer) _mbscpy:
  Does not check for buffer overflows when copying to
destination.
  Consider using a function version that stops copying at the end
of the
  buffer.
./test.c:71  [4] (buffer) lstrcat:
  Does not check for buffer overflows when concatenating to
destination.
./test.c:115  [4] (format) fprintf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:121  [4] (format) fprintf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:138  [4] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
./test.c:143  [4] (format) sprintf:
  Potential format string problem. Make format string constant.
./test.c:148  [4] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
./test.c:154  [4] (format) sprintf:
  Potential format string problem. Make format string constant.
./test.c:172  [4] (format) sprintf:
  Potential format string problem. Make format string constant.
./test.c:180  [4] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
./test.c:185  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:190  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:195  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:206  [4] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
./test.c:207  [4] (format) syslog:
  If syslog's format strings can be influenced by an attacker,
they can
  be exploited. Use a constant format string for syslog.
./test.c:213  [4] (format) snprintf:
  If format strings can be influenced by an attacker, they can be
  exploited, and note that sprintf variations do not always \0-
terminate. Use a   constant for the format specification.
./test.c:213  [4] (format) fprintf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:223  [4] (format) fprintf:
```

```
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
./test.c:94  [3] (shell) CreateProcess:
  This causes a new process to execute and is difficult to use
safely.
  Specify the application path in the first argument, NOT as part
of the
  second, or embedded spaces could allow an attacker to force a
different
  program to run.
./test.c:94  [3] (shell) CreateProcess:
  This causes a new process to execute and is difficult to use
safely.
  Specify the application path in the first argument, NOT as part
of the
  second, or embedded spaces could allow an attacker to force a
different
  program to run.
./test.c:100  [3] (buffer) getopt_long:
  Some older implementations do not protect against internal
buffer
  overflows . Check implementation on installation, or limit the
size of all
  string inputs.
./test.c:31  [2] (buffer) strcpy:
  Does not check for buffer overflows when copying to
destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily
misused). Risk
  is low because the source is a constant string.
./test.c:36  [2] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
Risk
  is low because the source has a constant maximum length.
./test.c:64  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds
checking,
  use functions that limit length, or ensure that the size is
larger than
  the maximum possible length.
./test.c:65  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds
checking,
  use functions that limit length, or ensure that the size is
larger than
  the maximum possible length.
./test.c:69  [2] (buffer) memcpy:
  Does not check for buffer overflows when copying to
destination. Make
  sure destination can always hold the source data.
./test.c:70  [2] (buffer) CopyMemory:
  Does not check for buffer overflows when copying to
destination. Make
  sure destination can always hold the source data.
./test.c:106  [2] (misc) fopen:
  Check when opening files - can an attacker redirect it (via
symlinks),
```

```
  force the opening of special file type (e.g., device files),
move
  things around to create a race condition, control its
ancestors, or change
  its contents?.
./test.c:131  [2] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
Risk
  is low because the source has a constant maximum length.
./test.c:154  [2] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
Risk
  is low because the source has a constant maximum length.
./test.c:30  [1] (buffer) strcpy:
  Does not check for buffer overflows when copying to
destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily
misused). Risk
  is low because the source is a constant character.
./test.c:35  [1] (buffer) sprintf:
  Does not check for buffer overflows. Use snprintf or vsnprintf.
Risk
  is low because the source is a constant character.
./test.c:45  [1] (buffer) scanf:
  it's unclear if the %s limit in the format string is small
enough.
  Check that the limit is sufficiently small, or use a different
input
  function.
./test.c:72  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check
for
  invalid pointers.
./test.c:73  [1] (buffer) _tcsncpy:
  Easily used incorrectly; doesn't always \0-terminate or check
for
  invalid pointers.
./test.c:74  [1] (buffer) strncat:
  Easily used incorrectly; doesn't always \0-terminate or check
for
  invalid pointers.
./test.c:77  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated (it could
cause a
  crash if unprotected).
./test.c:83  [1] (buffer) MultiByteToWideChar:
  Requires maximum length in CHARACTERS, not bytes. Risk is very
low,
  the length appears to be in characters not bytes.
./test.c:85  [1] (buffer) MultiByteToWideChar:
  Requires maximum length in CHARACTERS, not bytes. Risk is very
low,
  the length appears to be in characters not bytes.
./test.c:160  [1] (port) snprintf:
  On some very old systems, snprintf is incorrectly implemented
and
```

```
  permits buffer overflows; there are also incompatible standard
definitions
  of it. Check it during installation, or use something else.
./test.c:165  [1] (port) snprintf:
  On some very old systems, snprintf is incorrectly implemented
and
  permits buffer overflows; there are also incompatible standard
definitions
  of it. Check it during installation, or use something else.
Number of hits = 56
Number of Lines Analyzed = 228 in 0.96 seconds (498 lines/second)
2 hit(s) suppressed; use --neverignore to show them.
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Figure 4: RATS results for test.c

```
Entries in perl database: 33
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55


Analyzing test.c

RATS results.

Severity: High
Issue: gettext
Environment variables are highly untrustable input. They may be
of any length, and contain any data. Do not make any assumptions
regarding content or length. If at all possible avoid using them,
and if it is necessary, sanitize them and truncate them to a
reasonable length. gettext() can utilize the LC_ALL or
LC_MESSAGES environment variables.
      File: test.c
      Lines: 31 38
Severity: High
Issue: strcpy
Check to be sure that argument 2 passed to this function call
will not copy more data than can be handled, resulting in a
buffer overflow.
      File: test.c
      Lines: 31 34
Severity: High
Issue: sprintf
Check to be sure that the non-constant format string passed as
argument 2 to this function call does not come from an untrusted
source that could have added formatting characters that the code
is not prepared to handle.
      File: test.c
      Lines: 38 39 143 154 172
Severity: High
Issue: sprintf
Check to be sure that the format string passed as argument 2 to
```

this function call does not come from an untrusted source that could have added formatting characters that the code is not prepared to handle. Additionally, the format string could contain `%s' without precision that could result in a buffer overflow.
File: **test.c**
Lines: 38 39 143 154 172

Severity: High
Issue: printf
Check to be sure that the non-constant format string passed as argument 1 to this function call does not come from an untrusted source that could have added formatting characters that the code is not prepared to handle.
File: **test.c**
Lines: 42 185 190 195

Severity: High
Issue: scanf
Check to be sure that the format string passed as argument 2 to this function call does not come from an untrusted source that could have added formatting characters that the code is not prepared to handle. Additionally, the format string could contain `%s' without precision that could result in a buffer overflow.
File: **test.c**
Lines: 43 44 45 46

Severity: High
Issue: gets
Gets is unsafe!! No bounds checking is performed, buffer is easily overflowable by user. Use fgets(buf, size, stdin) instead.
File: **test.c**
Lines: 47 50 51

Severity: High
Issue: syslog
Truncate all input strings to a reasonable length before passing them to this function
File: **test.c**
Lines: 54 55 57 207

Severity: High
Issue: fixed size global buffer
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.
File: **test.c**
Lines: 64 65

Severity: High
Issue: _mbscpy
Check to be sure that argument 2 passed to this function call will not copy more data than can be handled, resulting in a buffer overflow.
File: **test.c**
Lines: 68

Severity: High
Issue: lstrcat
Check to be sure that argument 2 passed to this function call will not copy more data than can be handled, resulting in a buffer overflow.
File: **test.c**
Lines: 71

```
Severity: High
Issue: strncat
Consider using strlcat() instead.
      File: test.c
      Lines: 74 75
Severity: High
Issue: strncat
Check to be sure that argument 1 passed to this function call
will not copy more data than can be handled, resulting in a
buffer overflow.
      File: test.c
      Lines: 74 75
Severity: High
Issue: CreateProcess
Many program execution commands under Windows will search the
path for a program if you do not explicitly specify a full path
to the file. This can allow trojans to be executed instead. Also,
be sure to specify a file extension, since otherwise multiple
extensions will be tried by the operating system, providing
another opportunity for trojans.
      File: test.c
      Lines: 94
Severity: High
Issue: getopt_long
Truncate all input strings to a reasonable length before passing
them to this function
      File: test.c
      Lines: 100
Severity: High
Issue: fprintf
Check to be sure that the non-constant format string passed as
argument 2 to this function call does not come from an untrusted
source that could have added formatting characters that the code
is not prepared to handle.
      File: test.c
      Lines: 115 121
Severity: Medium
Issue: SetSecurityDescriptorDacl
If the third argument, pDacl, is NULL there is no protection from
attack. As an example, an attacker could set a Deny All to
Everyone ACE on such an object.
      File: test.c
      Lines: 92

Inputs detected at the following points


Total lines analyzed: 229
Total time 0.029155 seconds
7854 lines per second
```

Figure 5: Splint Results for test.c

```
Splint 3.0.1.6 --- 27 Mar 2002
```

```
test2.c:210:46: Comment starts inside comment
  A comment open sequence (/*) appears within a comment.  This
usually means an   earlier comment was not closed. (Use -
nestcomment to inhibit warning)
test2.c: (in function main)
test2.c:23:2: Path with no return in function declared to return
int
  There is a path through a function declared to return a value
on which there
  is no return statement. This means the execution may fall
through without
  returning a meaningful result to the caller. (Use -noret to
inhibit warning)
test2.c: (in function demo)
test2.c:31:12: Unrecognized identifier: gettext
  Identifier used in code has not been declared. (Use -unrecog to
inhibit
  warning)
test2.c:34:9: Parameter 1 (b) to function strcpy is declared
unique but may be
                aliased externally by parameter 2 (a)
  A unique or only parameter may be aliased by some other
parameter or visible
  global. (Use -mayaliasunique to inhibit warning)
test2.c:35:10: Unrecognized identifier: s
test2.c:37:25: Unrecognized identifier: bug
test2.c:38:2: Format string parameter to sprintf is not a
compile-time
                constant: gettext("hello %s")
  Format parameter is not known at compile-time.  This can lead
to security
  vulnerabilities because the arguments cannot be type checked.
(Use
  -formatconst to inhibit warning)
test2.c:39:13: Unrecognized identifier: unknown
test2.c:39:2: Format string parameter to sprintf is not a
compile-time
                co nstant: unknown
test2.c:42:9: Unrecognized identifier: bf
test2.c:42:13: Unrecognized identifier: x
test2.c:42:2: Format string parameter to printf is not a compile-
time constant:                  bf
test2.c:43:2: Return value (type int) ignored: scanf("%d", &x)
  Result returned by function call is not used. If this is
intended, can cast
  result to (void) to eliminate message. (Use -retvalint to
inhibit warning)
test2.c:44:2: Return value (type int) ignored: scanf("%s", s)
test2.c:45:2: Return value (type int) ignored: scanf("%10s", s)
test2.c:46:2: Return value (type int) ignored: scanf("%s", s)
test2.c:47:2: Use of gets leads to a buffer overflow
vulnerability.  Use fgets
                instead: gets
  Use of function that may lead to buffer overflow. (Use -
bufferoverflowhigh to   inhibit warning)
test2.c:47:7: Unrecognized identifier: f
test2.c:47:2: Return value (type char *) ignored: gets(f)
```

```
  Result returned by function call is not used. If this is
intended, can cast
  result to (void) to eliminate message. (Use -retvalother to
inhibit warning)
test2.c:50:2: Use of gets leads to a buffer overflow
vulnerability.  Use fgets
                 instead: gets
test2.c:50:2: Return value (type char *) ignored: gets(f)
test2.c:51:2: Use of gets leads to a buffer overflow
vulnerability.  Use fgets
                 instead: gets
test2.c:51:2: Return value (type char *) ignored: gets(f)
test2.c:54:2: Path with no return in function declared to return
int
test2.c: (in function demo2)
test2.c:63:3: Unrecognized identifier: _mbscpy
test2.c:64:3: Function memcpy called with 2 args, expects 3
  Types are incompatible. (Use -type to inhibit warning)
test2.c:64:12: Passed storage s not completely defined (*s is
undefined):
                  memcpy (..., s)
  Storage derivable from a parameter, return value or global is
not defined.
  Use /*@out@*/ to denote passed or returned storage which need
not be defined.   (Use -compdef to inhibit warning)
test2.c:65:3: Unrecognized identifier: CopyMemory
test2.c:66:3: Unrecognized identifier: lstrcat
test2.c:67:3: Function strncpy called with 2 args, expects 3
test2.c:68:3: Unrecognized identifier: _tcsncpy
test2.c:71:3: Unrecognized identifier: _tcsncat
test2.c:72:3: Assignment of size_t to int: n = strlen(d)
  To allow arbitrary integral types to match any integral type,
use
  +matchanyintegral.
test2.c:74:3: Unrecognized identifier: MultiByteToWideChar
test2.c:74:23: Unrecognized identifier: CP_ACP
test2.c:74:32: Unrecognized identifier: szName
test2.c:74:42: Unrecognized identifier: wszUserName
test2.c:87:3: Unrecognized identifier: SetSecurityDescriptorDacl
test2.c:87:30: Unrecognized identifier: sd
test2.c:89:3: Unrecognized identifier: CreateProcess
test2.c:90:2: Path with no return in function declared to return
int
test2.c: (in function getopt_example)
test2.c:95:13: Unrecognized identifier: optc
test2.c:95:20: Unrecognized identifier: getopt_long
test2.c:95:49: Unrecognized identifier: longopts
test2.c:97:2: Path with no return in function declared to return
int
test2.c: (in function testfile)
test2.c:102:10: Possibly null storage f passed as non-null param:
fclose (f)
  A possibly null pointer is passed as a parameter corresponding
to a formal
  parameter with no /*@null@*/ annotation.  If NULL may be used
for this
```

```
  parameter, add a /*@null@*/ annotation to the function
parameter declaration.   (Use -nullpass to inhibit warning)
   test2.c:101:7: Storage f may become null
test2.c:102:3: Return value (type int) ignored: fclose(f)
test2.c:103:2: Path with no return in function declared to return
int
test2.c: (in function demo3)
test2.c:112:17: Unrecognized identifier: variable
test2.c:112:1: Format string parameter to fprintf is not a
compile-time
                  constant: variable
test2.c:118:17: Unrecognized identifier: format
test2.c:118:25: Unrecognized identifier: variable1
test2.c:118:36: Unrecognized identifier: variable2
test2.c:118:1: Format string parameter to fprintf is not a
compile-time
                  constant: format
test2.c:128:9: Unrecognized identifier: buffer
test2.c:135:1: No argument corresponding to sprintf format code 2
(%s): "%s"
   test2.c:135:19: Corresponding format code
test2.c:140:1: Format string parameter to sprintf is not a
compile-time
                  constant: variable
test2.c:151:31: Unrecognized identifier: buffer1
test2.c:151:23: Format string parameter to sprintf is not a
compile-time
                   constant: variable
test2.c:157:1: Unrecognized identifier: snprintf
test2.c:157:66: Unrecognized identifier: filename
test2.c:169:1: Format string parameter to sprintf is not a
compile-time
                  constant: variable
test2.c:177:29: Unrecognized identifier: one
test2.c:177:34: Unrecognized identifier: two
test2.c:177:39: Unrecognized identifier: three
test2.c:182:35: Unrecognized identifier: string
test2.c:182:1: Format string parameter to printf is not a
compile-time
                  constant: (variable ? "%4" : "%3s")
test2.c:187:20: Unrecognized identifier: fmt1
test2.c:187:27: Unrecognized identifier: fmt2
test2.c:187:34: Unrecognized identifier: string3
test2.c:187:1: Format string parameter to printf is not a
compile-time
                  constant: (variable ? fmt1 : fmt2)
test2.c:192:20: Unrecognized identifier: string1
test2.c:192:30: Unrecognized identifier: string2
test2.c:192:1: Format string parameter to printf is not a
compile-time
                  constant: (variable ? string1 : string2)
test2.c:203:43: Unrecognized identifier: i
test2.c:203:46: Unrecognized identifier: remoteident
test2.c:204:1: Unrecognized identifier: syslog
test2.c:204:8: Unrecognized identifier: LOG_DEBUG
test2.c:220:1: Array fetch from non-array ([function (FILE *,
char *, ...)
```

```
                  returns int]): fprintf[1]
test2.c:220:14: Statement has no effect: <error> = 1
  Statement has no visible effect --- no values are modified.
(Use -noeffect to
  inhibit warning)
test2.c:225:1: Unrecognized identifier: err
test2.c:227:2: Path with no return in function declared to return
int

Finished checking --- 82 code warnings
```

Figure 6: ESC/Java results for telnet.java

```
ESC/Java version 1.2.4, 27 September 2001
Error: I/O error: modules (Access is denied)
Error: I/O error: -v (The system cannot find the file specified)

telnet ...

telnet: getAppletInfo() ...
------------------------------------------------------------------
-------
telnet.java:128: Warning: Possible null dereference (Null)
    info += "Terminal emulation: "+term.getTerminalType()+
                                   ^
------------------------------------------------------------------
-------
telnet.java:130: Warning: Possible null dereference (Null)
    info += "Terminal IO version: "+tio.toString()+"\n";
                                   ^
------------------------------------------------------------------
-------
telnet.java:134: Warning: Possible null dereference (Null)
        info += "   + "+(modules.elementAt(i)).toString()+"\n";
                                               ^
Execution trace information:
    Executed then branch in "telnet.java", line 131, col 46.
    Reached top of loop after 0 iterations in "telnet.java", line
133, col 6.

------------------------------------------------------------------
-------
    [0.51 s]  failed

telnet: getParameterInfo() ...
------------------------------------------------------------------
-------
telnet.java:161: Warning: Possible null dereference (Null)
    System.arraycopy(tinfo, 0, pinfo, 3, tinfo.length);
                                         ^
Execution trace information:
    Executed then branch in "telnet.java", line 157, col 39.
    Executed else branch in "telnet.java", line 159, col 9.

------------------------------------------------------------------
-------
```

```
    [0.441 s]  failed

telnet: getParameter(java.lang.String) ...
------------------------------------------------------------------
-------
telnet.java:175: Warning: Possible type cast error (Cast)
    return (String)params.get(name);
           ^
Execution trace information:
    Executed else branch in "telnet.java", line 174, col 4.

------------------------------------------------------------------
-------
    [0.1 s]  failed

telnet: main(java.lang.String[]) ...
------------------------------------------------------------------
-------
telnet.java:188: Warning: Possible null dereference (Null)
    switch(args.length)
               ^
------------------------------------------------------------------
-------
    [0.241 s]  failed

telnet: init() ...
------------------------------------------------------------------
-------
telnet.java:234: Warning: Possible null dereference (Null)
      address = getDocumentBase().getHost();
                                  ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.

------------------------------------------------------------------
-------
telnet.java:252: Warning: Possible null dereference (Null)
      term =
(Terminal)Class.forName("display."+emulation).newInstance ...
                                                          ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.

------------------------------------------------------------------
-------
telnet.java:252: Warning: Possible type cast error (Cast)
      term =
(Terminal)Class.forName("display."+emulation).newInstance ...
            ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
```

```
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.

-------------------------------------------------------------------
-------
telnet.java:271: Warning: Possible null dereference (Null)
        initFile = tmp.substring(tmp.indexOf(','+1));
                                 ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.
    Reached top of loop after 0 iterations in "telnet.java", line
263, col 4.
    Executed then branch in "telnet.java", line 268, col 33.

-------------------------------------------------------------------
-------
telnet.java:279: Warning: Possible null dereference (Null)
      Object obj =
(Object)Class.forName("modules."+tmp).newInstance() ...
                                                    ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.
    Reached top of loop after 0 iterations in "telnet.java", line
263, col 4.
    Executed then branch in "telnet.java", line 268, col 33.
    Executed then branch in "telnet.java", line 275, col 33.

-------------------------------------------------------------------
-------
telnet.java:283: Warning: Possible type cast error (Cast)
        ((Module)obj).setLoader(this);
         ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.
    Reached top of loop after 0 iterations in "telnet.java", line
263, col 4.
    Executed then branch in "telnet.java", line 268, col 33.
    Executed then branch in "telnet.java", line 275, col 33.

-------------------------------------------------------------------
-------
telnet.java:283: Warning: Possible null dereference (Null)
        ((Module)obj).setLoader(this);
                  ^
Execution trace information:
```

```
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.
    Reached top of loop after 0 iterations in "telnet.java", line
263, col 4.
    Executed then branch in "telnet.java", line 268, col 33.
    Executed then branch in "telnet.java", line 275, col 33.

-------------------------------------------------------------------
-------
telnet.java:293: Warning: Possible type cast error (Cast)
        Component component = (Component)obj;
                              ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.
    Reached top of loop after 0 iterations in "telnet.java", line
263, col 4.
    Executed then branch in "telnet.java", line 268, col 33.
    Executed then branch in "telnet.java", line 275, col 33.

-------------------------------------------------------------------
-------
telnet.java:294: Warning: Possible null dereference (Null)
        if(position.equals("North")) {
                   ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed then branch in "telnet.java", line 237, col 6.
    Executed then branch in "telnet.java", line 242, col 6.
    Executed then branch in "telnet.java", line 243, col 8.
    Executed then branch in "telnet.java", line 248, col 6.
    Reached top of loop after 0 iterations in "telnet.java", line
263, col 4.
    Executed then branch in "telnet.java", line 268, col 33.
    Executed then branch in "telnet.java", line 275, col 33.

-------------------------------------------------------------------
-------
telnet.java:316: Warning: Possible unexpected exception
(Exception)
  }
  ^
Execution trace information:
    Executed then branch in "telnet.java", line 234, col 6.
    Executed else branch in "telnet.java", line 239, col 6.
    Routine call returned exceptionally in "telnet.java", line
239, col 21.

-------------------------------------------------------------------
-------
```

```
Caution: Not checking method init() of type telnet completely
because warning limit (PROVER_CC_LIMIT) reached
    [2.493 s]  failed

telnet: start() ...
    [0.02 s]  passed

telnet: stop() ...
    [0.01 s]  passed

telnet: run() ...
-------------------------------------------------------------------
-------
telnet.java:343: Warning: Possible null dereference (Null)
        String tmp = new String(tio.receive(), 0);
                                ^
Execution trace information:
    Reached top of loop after 0 iterations in "telnet.java", line
341, col 4.

-------------------------------------------------------------------
-------
telnet.java:348: Warning: Possible null dereference (Null)
          while(modlist.hasMoreElements()) {
                        ^
Execution trace information:
    Reached top of loop after 0 iterations in "telnet.java", line
341, col 4.
    Executed then branch in "telnet.java", line 346, col 28.
    Reached top of loop after 0 iterations in "telnet.java", line
348, col 10.

-------------------------------------------------------------------
-------
telnet.java:349: Warning: Possible type cast error (Cast)
            Module m = (Module)modlist.nextElement();
                       ^
Execution trace information:
    Reached top of loop after 0 iterations in "telnet.java", line
341, col 4.
    Executed then branch in "telnet.java", line 346, col 28.
    Reached top of loop after 0 iterations in "telnet.java", line
348, col 10.

-------------------------------------------------------------------
-------
telnet.java:350: Warning: Possible null dereference (Null)
            String modified = m.receive(tmp);
                              ^
Execution trace information:
    Reached top of loop after 0 iterations in "telnet.java", line
341, col 4.
    Executed then branch in "telnet.java", line 346, col 28.
    Reached top of loop after 0 iterations in "telnet.java", line
348, col 10.
```

```
------------------------------------------------------------------
-------
telnet.java:358: Warning: Possible null dereference (Null)
        term.putString(tmp);
            ^
Execution trace information:
    Reached top of loop after 0 iterations in "telnet.java", line
341, col 4.
    Executed then branch in "telnet.java", line 346, col 28.
    Reached top of loop after 0 iterations in "telnet.java", line
348, col 10.

------------------------------------------------------------------
-------
    [0.511 s]  failed

telnet: connect() ...
    [0.01 s]  passed

telnet: connect(java.lang.String) ...
    [0.02 s]  passed

telnet: connect(java.lang.String, int) ...
------------------------------------------------------------------
-------
telnet.java:408: Warning: Possible null dereference (Null)
      term.putString("Trying "+address+(port==23?"":" "+port)+"
...\n\ ...
          ^
Execution trace information:
    Executed else branch in "telnet.java", line 394, col 4.
    Executed else branch in "telnet.java", line 400, col 11.
    Executed then branch in "telnet.java", line 405, col 22.
    Executed then branch in "telnet.java", line 408, col 49.

------------------------------------------------------------------
-------
telnet.java:414: Warning: Possible attempt to allocate array of
negative length (NegSize)
          byte[] bytes = new byte[str.length()];
                                 ^
Execution trace information:
    Executed else branch in "telnet.java", line 394, col 4.
    Executed else branch in "telnet.java", line 400, col 11.
    Executed then branch in "telnet.java", line 405, col 22.
    Executed then branch in "telnet.java", line 408, col 49.
    Executed then branch in "telnet.java", line 411, col 26.

------------------------------------------------------------------
-------
telnet.java:424: Warning: Possible null dereference (Null)
             getParameter("localecho").equals("no")
                                      ^
Execution trace information:
    Executed else branch in "telnet.java", line 394, col 4.
    Executed else branch in "telnet.java", line 400, col 11.
    Executed then branch in "telnet.java", line 405, col 22.
```

```
    Executed then branch in "telnet.java", line 408, col 49.
    Executed then branch in "telnet.java", line 411, col 26.

-----------------------------------------------------------------
-------
telnet.java:431: Warning: Possible null dereference (Null)
          while(modlist.hasMoreElements())
                        ^
Execution trace information:
    Executed else branch in "telnet.java", line 394, col 4.
    Executed else branch in "telnet.java", line 400, col 11.
    Executed then branch in "telnet.java", line 405, col 22.
    Executed then branch in "telnet.java", line 408, col 49.
    Executed then branch in "telnet.java", line 411, col 26.
    Executed else branch in "telnet.java", line 423, col 1.
    Executed then branch in "telnet.java", line 429, col 28.
    Reached top of loop after 0 iterations in "telnet.java", line
431, col 10.

-----------------------------------------------------------------
-------
telnet.java:433: Warning: Possible type cast error (Cast)
            ((Module)modlist.nextElement()).connect(address,
port);
             ^
Execution trace information:
    Executed else branch in "telnet.java", line 394, col 4.
    Executed else branch in "telnet.java", line 400, col 11.
    Executed then branch in "telnet.java", line 405, col 22.
    Executed then branch in "telnet.java", line 408, col 49.
    Executed then branch in "telnet.java", line 411, col 26.
    Executed else branch in "telnet.java", line 423, col 1.
    Executed then branch in "telnet.java", line 429, col 28.
    Reached top of loop after 0 iterations in "telnet.java", line
431, col 10.

-----------------------------------------------------------------
-------
telnet.java:433: Warning: Possible null dereference (Null)
            ((Module)modlist.nextElement()).connect(address,
port);
                                            ^
Execution trace information:
    Executed else branch in "telnet.java", line 394, col 4.
    Executed else branch in "telnet.java", line 400, col 11.
    Executed then branch in "telnet.java", line 405, col 22.
    Executed then branch in "telnet.java", line 408, col 49.
    Executed then branch in "telnet.java", line 411, col 26.
    Executed else branch in "telnet.java", line 423, col 1.
    Executed then branch in "telnet.java", line 429, col 28.
    Reached top of loop after 0 iterations in "telnet.java", line
431, col 10.

-----------------------------------------------------------------
-------
    [1.402 s]  failed
```

```
telnet: disconnect() ...
-------------------------------------------------------------------
-------
telnet.java:471: Warning: Possible null dereference (Null)
        while(modlist.hasMoreElements())
                      ^
Execution trace information:
    Executed else branch in "telnet.java", line 462, col 4.
    Executed then branch in "telnet.java", line 469, col 26.
    Reached top of loop after 0 iterations in "telnet.java", line
471, col 8.

-------------------------------------------------------------------
-------
telnet.java:473: Warning: Possible type cast error (Cast)
          ((Module)modlist.nextElement()).disconnect();
           ^
Execution trace information:
    Executed else branch in "telnet.java", line 462, col 4.
    Executed then branch in "telnet.java", line 469, col 26.
    Reached top of loop after 0 iterations in "telnet.java", line
471, col 8.

-------------------------------------------------------------------
-------
telnet.java:473: Warning: Possible null dereference (Null)
          ((Module)modlist.nextElement()).disconnect();
                                          ^
Execution trace information:
    Executed else branch in "telnet.java", line 462, col 4.
    Executed then branch in "telnet.java", line 469, col 26.
    Reached top of loop after 0 iterations in "telnet.java", line
471, col 8.

-------------------------------------------------------------------
-------
telnet.java:475: Warning: Possible null dereference (Null)
      term.putString("\n\rConnection closed.\n\r");
          ^
Execution trace information:
    Executed else branch in "telnet.java", line 462, col 4.
    Executed then branch in "telnet.java", line 469, col 26.
    Reached top of loop after 0 iterations in "telnet.java", line
471, col 8.

-------------------------------------------------------------------
-------
    [0.401 s]  failed

telnet: send(java.lang.String) ...
-------------------------------------------------------------------
-------
telnet.java:495: Warning: Possible null dereference (Null)
      byte[] bytes = new byte[str.length()];
                             ^
Execution trace information:
    Executed then branch in "telnet.java", line 494, col 18.
```

```
------------------------------------------------------------------
-------
telnet.java:495: Warning: Possible attempt to allocate array of
negative length (NegSize)
      byte[] bytes = new byte[str.length()];
                             ^
Execution trace information:
    Executed then branch in "telnet.java", line 494, col 18.

------------------------------------------------------------------
-------
telnet.java:497: Warning: Possible null dereference (Null)
      tio.send(bytes);
         ^
Execution trace information:
    Executed then branch in "telnet.java", line 494, col 18.

------------------------------------------------------------------
-------
telnet.java:500: Warning: Possible null dereference (Null)
          term.putString("\r\n");
               ^
Execution trace information:
    Executed then branch in "telnet.java", line 494, col 18.
    Executed then branch in "telnet.java", line 498, col 20.
    Executed then branch in "telnet.java", line 500, col 10.

------------------------------------------------------------------
-------
telnet.java:502: Warning: Possible null dereference (Null)
          term.putString(str);
               ^
Execution trace information:
    Executed then branch in "telnet.java", line 494, col 18.
    Executed then branch in "telnet.java", line 498, col 20.
    Executed else branch in "telnet.java", line 502, col 10.

------------------------------------------------------------------
-------
    [0.41 s]  failed

telnet: writeToSocket(java.lang.String) ...
------------------------------------------------------------------
-------
telnet.java:522: Warning: Possible null dereference (Null)
        byte[] bytes = new byte[str.length()];
                                ^
Execution trace information:
    Executed then branch in "telnet.java", line 521, col 21.

------------------------------------------------------------------
-------
telnet.java:522: Warning: Possible attempt to allocate array of
negative length (NegSize)
        byte[] bytes = new byte[str.length()];
                               ^
```

```
Execution trace information:
    Executed then branch in "telnet.java", line 521, col 21.

------------------------------------------------------------------
-------
telnet.java:524: Warning: Possible null dereference (Null)
        tio.send(bytes);
           ^
Execution trace information:
    Executed then branch in "telnet.java", line 521, col 21.

------------------------------------------------------------------
-------
    [0.291 s]  failed

telnet: writeToUser(java.lang.String) ...
    [0.02 s]  passed

telnet: notifyStatus(java.util.Vector) ...
------------------------------------------------------------------
-------
telnet.java:553: Warning: Possible null dereference (Null)
    String what = (String)status.elementAt(0);
                          ^
------------------------------------------------------------------
-------
telnet.java:553: Warning: Possible type cast error (Cast)
    String what = (String)status.elementAt(0);
                  ^
------------------------------------------------------------------
-------
telnet.java:555: Warning: Possible null dereference (Null)
    if(what.equals("NAWS"))
           ^
------------------------------------------------------------------
-------
telnet.java:556: Warning: Possible null dereference (Null)
      return term.getSize();
                  ^
Execution trace information:
    Executed then branch in "telnet.java", line 556, col 6.

------------------------------------------------------------------
-------
telnet.java:558: Warning: Possible null dereference (Null)
    if(term.getTerminalType() == null)
             ^
Execution trace information:
    Executed else branch in "telnet.java", line 555, col 4.
    Executed then branch in "telnet.java", line 558, col 6.

------------------------------------------------------------------
-------
telnet.java:569: Warning: Possible null dereference (Null)
          getParameter("localecho").equals("auto")
                                   ^
Execution trace information:
```

```
    Executed else branch in "telnet.java", line 555, col 4.
    Executed else branch in "telnet.java", line 557, col 4.
    Short circuited boolean operation in "telnet.java", line 562,
col 32.
    Executed then branch in "telnet.java", line 562, col 63.
    Executed then branch in "telnet.java", line 565, col 1.
    Executed then branch in "telnet.java", line 567, col 1.

------------------------------------------------------------------
-------
    [0.44 s]  failed

telnet: telnet() ...
    [0.02 s]  passed
  [9.043 s total]
1 caution
45 warnings
2 errors
```

Figure 7: hello.c

This code is courtesy of Hao Chen and David Wagner, creators of MOPS.

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <pwd.h>
5  _
6  void drop_priv()
7  {
8  struct passwd *passwd;
9  _
10 if ((passwd = getpwuid(getuid())) == NULL)
11 {
12 printf("getpwuid() failed");
13 return;
14 }
15 printf("Drop user %s's privilege\n", passwd->pw_name);
16 seteuid(getuid());
17 }
18 _
19 int main(int argc, char *argv[])
20 {
21 drop_priv();
22 printf("About to exec\n");
23 execv(argv[1], argv + 1);
24 }
```

Figure 8: hello.tra

This is produced by running a modelcheck with MOPS.

```
-*- compilation -*-
Trace from hello.s.tra
hello.c:19: <euid_0,before_exec>  1
hello.c:21: <euid_0,before_exec>  1
hello.c:6: <euid_0,before_exec>  2
hello.c:8: <euid_0,before_exec>  2
hello.c:10: <euid_0,before_exec>  2
hello.c:10: <euid_0,before_exec>  2
hello.c:10: <euid_0,before_exec>  2
hello.c:10: <euid_0,before_exec>  2
hello.c:12: <euid_0,before_exec>  2
hello.c:12: <euid_0,before_exec>  2
hello.c:13: <euid_0,before_exec>  2
hello.c:6: <euid_0,before_exec>  2
hello.c:22: <euid_0,before_exec>  1
hello.c:22: <euid_0,before_exec>  1
hello.c:23: <euid_0,before_exec>  1
hello.c:23: <euid_0,before_exec>  1
hello.c:23: <euid_0,before_exec>  1
hello.c:23: <euid_0,before_exec>  1
hello.c:23: <euid_0,before_exec>  1
hello.c:23: <euid_0,before_exec>  1
hello.c:23: <euid_0,after_exec>  1
```

Resources

1) Clark, James. "'expat - XML Parser Toolkit' Homepage" , 2000 http://www.jclark.com/xml/expat.html

2) Newsham, Tim. "Format String Attacks", September 2000 http://online.securityfocus.com/guest/3342

3) Wheeler, David A. "Write It Secure: Format Strings and Locale Filtering", 2000. http://www.dwheeler.com/essays/write_it_secure_1.html

4) "NCSA Secure Programming Guidelines". http://archive.ncsa.uiuc.edu/General/Grid/ACES/security/programming/

5) Halloway, Stewart. "Controlling Package Access with Security Permissions", January 30, 2001. http://developer.java.sun.com/developer/JDCTechTips/2001/tt0130.html

6) Halloway, Stewart. "Using Security Manager", September 26, 2000. http://developer.java.sun.com/developer/TechTips/2000/tt0926.html#tip1

7) Chen, Hao & Wagner, David. "MOPS – MOdelchecking Programs for Security Properties", 2002. http://www.cs.berkeley.edu/~daw/mops/

8) Beattie, Malcolm. "Perl Safe Module Documentation." They Dot Com. http://www.they.com/doc/local/perl/lib/Safe.html

9) "perlsec." Perldoc.com. http://www.perldoc.com/perl5.6/pod/perlsec.html

10) Wall, Larry & Christiansen, Tom & Schwartz, Randal L.. Programming Perl, 2nd Edition. Reading: O'Reilly and Associates, Inc., September 1996.

11) DeKok, Alan. "PScan: A limited problem scanner for C source files", July 7, 2000. http://www.striker.ottawa.on.ca/~aland/pscan/

12) Raynal, Frederic. "Avoiding security holes when developing an application - 5: race conditions", September 18, 2002. http://www.security-labs.org/index.php3?page=122

13) Stein, Lincoln D. & Stewart, John N. "The World Wide Web Security FAQ", February 4, 2002.

http://www.w3.org/Security/faq/www-security-faq.html

14) Volatile. "Setuid/Setgid Tutorial." New Order, January 15, 2002. http://neworder.box.sk/newsread.php?newsid=2380

15) Al-herbish, Thamer & Roozemaal, Peter. "Secure Unix Programming FAQ", May 16, 1999. http://www.whitefang.com/sup/secure-faq.html

16) Wheeler, David A. "'Flawfinder' Homepage". http://www.dwheeler.com/flawfinder/

17) Evans, David. "'Splint' Homepage". University of Virginia, Department of Computer Science, 2002. http://splint.org/

18) "Extended Static Checking for Java." Compaq Computer Corporation, 2000. http://www.research.compaq.com/SRC/esc/Esc.html

19) "'RATS' Homepage." Secure Software. http://www.securesoftware.com/rats.php

Last Updated: July 20th, 2018

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Riyadh July 2018	Riyadh, SA	Jul 28, 2018 - Aug 02, 2018	Live Event
SANS Pittsburgh 2018	Pittsburgh, PAUS	Jul 30, 2018 - Aug 04, 2018	Live Event
Security Operations Summit & Training 2018	New Orleans, LAUS	Jul 30, 2018 - Aug 06, 2018	Live Event
SANS Hyderabad 2018	Hyderabad, IN	Aug 06, 2018 - Aug 11, 2018	Live Event
Security Awareness Summit & Training 2018	Charleston, SCUS	Aug 06, 2018 - Aug 15, 2018	Live Event
SANS Boston Summer 2018	Boston, MAUS	Aug 06, 2018 - Aug 11, 2018	Live Event
SANS San Antonio 2018	San Antonio, TXUS	Aug 06, 2018 - Aug 11, 2018	Live Event
SANS August Sydney 2018	Sydney, AU	Aug 06, 2018 - Aug 25, 2018	Live Event
SANS New York City Summer 2018	New York City, NYUS	Aug 13, 2018 - Aug 18, 2018	Live Event
SANS Northern Virginia- Alexandria 2018	Alexandria, VAUS	Aug 13, 2018 - Aug 18, 2018	Live Event
SANS Krakow 2018	Krakow, PL	Aug 20, 2018 - Aug 25, 2018	Live Event
Data Breach Summit & Training 2018	New York City, NYUS	Aug 20, 2018 - Aug 27, 2018	Live Event
SANS Chicago 2018	Chicago, ILUS	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Prague 2018	Prague, CZ	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Virginia Beach 2018	Virginia Beach, VAUS	Aug 20, 2018 - Aug 31, 2018	Live Event
SANS San Francisco Summer 2018	San Francisco, CAUS	Aug 26, 2018 - Aug 31, 2018	Live Event
SANS Copenhagen August 2018	Copenhagen, DK	Aug 27, 2018 - Sep 01, 2018	Live Event
SANS SEC504 @ Bangalore 2018	Bangalore, IN	Aug 27, 2018 - Sep 01, 2018	Live Event
SANS Wellington 2018	Wellington, NZ	Sep 03, 2018 - Sep 08, 2018	Live Event
SANS Amsterdam September 2018	Amsterdam, NL	Sep 03, 2018 - Sep 08, 2018	Live Event
SANS Tokyo Autumn 2018	Tokyo, JP	Sep 03, 2018 - Sep 15, 2018	Live Event
SANS Tampa-Clearwater 2018	Tampa, FLUS	Sep 04, 2018 - Sep 09, 2018	Live Event
SANS MGT516 Beta One 2018	Arlington, VAUS	Sep 04, 2018 - Sep 08, 2018	Live Event
Threat Hunting & Incident Response Summit & Training 2018	New Orleans, LAUS	Sep 06, 2018 - Sep 13, 2018	Live Event
SANS Baltimore Fall 2018	Baltimore, MDUS	Sep 08, 2018 - Sep 15, 2018	Live Event
SANS Alaska Summit & Training 2018	Anchorage, AKUS	Sep 10, 2018 - Sep 15, 2018	Live Event
SANS Munich September 2018	Munich, DE	Sep 16, 2018 - Sep 22, 2018	Live Event
SANS London September 2018	London, GB	Sep 17, 2018 - Sep 22, 2018	Live Event
SANS Network Security 2018	Las Vegas, NVUS	Sep 23, 2018 - Sep 30, 2018	Live Event
SANS DFIR Prague Summit & Training 2018	Prague, CZ	Oct 01, 2018 - Oct 07, 2018	Live Event
Oil & Gas Cybersecurity Summit & Training 2018	Houston, TXUS	Oct 01, 2018 - Oct 06, 2018	Live Event
SANS Brussels October 2018	Brussels, BE	Oct 08, 2018 - Oct 13, 2018	Live Event
SANS Pen Test Berlin 2018	OnlineDE	Jul 23, 2018 - Jul 28, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced