GRAMMATECH

White Paper

Protecting Against Tainted Data in Embedded Apps with Static Analysis

Attacks against embedded systems are growing in frequency as malicious hackers become more sophisticated in their methods. These vulnerabilities are being exploited by hostile users to gain access to a system so they may subvert its use. These exploits are typically triggered when a hostile user sends data over an input channel.

Programmers can defend against these defects by treating input data as potentially hazardous (tainted) and carefully checking the data for validity before use. It is challenging, though, to find these locations because it requires manually tracing the flow of data through the program. This paper describes a static analysis technique – taint analysis – that can be used to find how potentially hazardous inputs can flow through a program to reach sensitive parts of code, empowering developers to identify and eliminate these dangerous vulnerabilities much more effectively.

Background

As embedded applications become more feature-rich, the risks of security vulnerabilities are increasing. For example, electronic systems in automobiles have been proven to be at particular risk because cars are an especially juicy target for attackers. Researchers demonstrated recently that it was relatively easy to find software security vulnerabilities in a late-model car and exploit them successfully to remotely unlock the doors and even start the engine.

Until recently, hackers had not been frequently targeting embedded systems because doing so required physical access to the device. However, with the accelerating network connectivity of embedded devices, malicious hackers have a new, virtual attack path. Despite the trend toward greater device connectivity, awareness of the risks of insecure code is still low among embedded developers.

In the parlance of secure programming, unchecked input values are said to be **tainted**. Tainted data vulnerabilities should always be a concern for developers, even when security is not as important because tainted data values also cause quality issues including unexpected device behavior and system crashes. Any software that reads input from any type of sensor should treat all values from the sensor as potentially dangerous. The values might be out-of-range due to a hardware failure, and if the program is not prepared to check the values, then they might cause the software to crash later. The same techniques that defend against security vulnerabilities can also be used to defend against rogue data values, so taint analysis techniques are also effective at finding and improving the quality of the most risky parts of the code.

Systems that are comprised of code supplied by several different vendors, or sources such as open-source code or code from trusted third parties, are at particular risk. This is because research has proven that security vulnerabilities proliferate at the boundaries between code components, often due to innocent disagreements in interpretation of the interface specifications.

Programmers can defend against such defects by treating inputs from potentially risky channels as hazardous until the validity of the data has been checked. It can be difficult to check that a program handles tainted data properly because doing so involves tracking its flow through the structure of the code. This can be tedious, even for relatively small programs, and for most real-world applications, it is infeasible to do manually.

Exploiting Programming Errors with Tainted Data

The biggest risk of using values read from an unverified channel is that an attacker can use the channel to trigger security vulnerabilities or cause the program to crash. Many types of issues can be triggered by tainted data, including buffer overruns, SQL injection, command injection, cross-site scripting, arithmetic overflow, and path traversal. (For more details on these and other classes of defect, see the Common Weakness Enumeration at http://cwe.mitre.org.) Many of the most damaging cyber-attacks in the last two decades have been caused by the infamous buffer overrun. As this is such a pervasive vulnerability, and because it illustrates the importance of taint analysis, it is worth explaining in some detail.

What is Tainted Data?

TAINT SOURCES, SINKS, AND CLEANSERS

In the context of taint analysis, a **taint source** is a location in the program where data is being read from a risky source. For instance, in the example on the next page, it is the call to getenv().

A **taint sink** is a location to which tainted data should not flow, unless it has been checked for validity, such as the call to strcpy() in the example.

Once a value has been checked, it is said to have been **cleansed** of the taint.

Programs take input from multiple sources, so the environment in which the program will execute determines the level of risk associated with each source.

Taint sources include the following:

- Environment variables
- File contents
- File metadata, such as a file's permissions or datestamps
- The network
- Network services, such as the results of a DNS query
- The system clock
- The registry as found on Windows systems

A PROGRAM'S ATTACK SURFACE

Any program may have other kinds of inputs that could potentially be hazardous. A program, for example, that reads input from a device with an infrared sensor should probably consider that channel as dangerous.

Security analysts define the points of exposure to a potentially hostile attacker as the program's **attack surface**. To assess a program's risk, it is useful to first gain an understanding of what its attack surface is, and this corresponds closely to the program's taint sources.

There are several ways in which a buffer overrun can be exploited by an attacker, but here we describe the classic case that makes it possible for the attacker to hijack the process and force it to run arbitrary code. In this example, the buffer is on the stack. Consider the following code:

```
void config(void)
{
    char buf[100];
    int count;
    …
    strcpy(buf, getenv(“CONFIG”));
    …
}
```

In this example, the input from the outside world is through a call to getenv that retrieves the value of the environment variable named “CONFIG”.

The programmer who wrote this code was expecting the value of the environment variable to fit in the buffer, but there is nothing that checks that this is so. If the attacker has control over the value of that environment variable, then assigning a value whose length exceeds 100 will cause a buffer overrun to occur. Because buf is an automatic variable, which will be placed on the stack as part of the activation record for the procedure, any characters after the first 100 will be written to the parts of the program stack beyond the boundaries of buf. The variable named count may be overwritten (depending on how the compiler chose to allocate space on the stack). If so, then the value of that variable is under the control of the attacker.

This is bad enough, but the real prize for the attacker is that the stack contains the address to which the program will jump once it has finished executing the procedure. To exploit this vulnerability, the attacker can set the value of the variable to a specially-crafted string that encodes a return address of his choosing. When the CPU gets to the end of the function, it will return to that address instead of the address of the function’s caller.

Now, if the code is executed in an environment where the attacker does not have control of the value of the environment variable, then it may be impossible to exploit this vulnerability. Nevertheless, the code is clearly very risky and remains a liability if left unfixed. A programmer might also be tempted to re-use this code in a different program where it may not be safe to run.

This example is taking its input from the environment, but the code would be just as risky if the string was being read from another input source, such as the file system or a network channel. The most risky input channels are those over which an attacker has control.

As mentioned, manually finding program errors that are sensitive to tainted values is extremely time-consuming, so automation is the best approach.

Automated Taint Analysis

Taint analysis is a form of static analysis. Before describing how it is done, we start with a brief introduction to the class of static analysis tools that are capable of implementing taint analysis.

Roughly speaking, advanced static-analysis tools work as follows. First, they must create a model of the entire program, which they do by reading and parsing each input file. The model consists of representations, such as abstract-syntax trees for each compilation unit, control-flow graphs for each subprogram, symbol-tables, and the call graph. Checkers that find defects are implemented in terms of various kinds of queries on those representations. Superficial bugs can be found by doing pattern matching on the abstract-syntax tree or the symbol tables.

THE THREE MAIN CATEGORIES OF DEFECTS

1 Bugs that violate the fundamental rules of the runtime, thereby causing the program’s behavior to be undefined.

These bugs include memory errors, such as null pointer dereferences and buffer overruns; concurrency errors, such as data races; and many other bugs, such as use of uninitialized memory.

2 Defects that arise because the program breaks the rules of using a standard API.

For example, the C library does not specify what happens when the same file descriptor is closed twice, and since this makes no sense to do deliberately, it is probably a bug. Leaks of finite resources, i.e. memory or file descriptors, also fall into this category.

3 Inconsistencies or contradictions in the code.

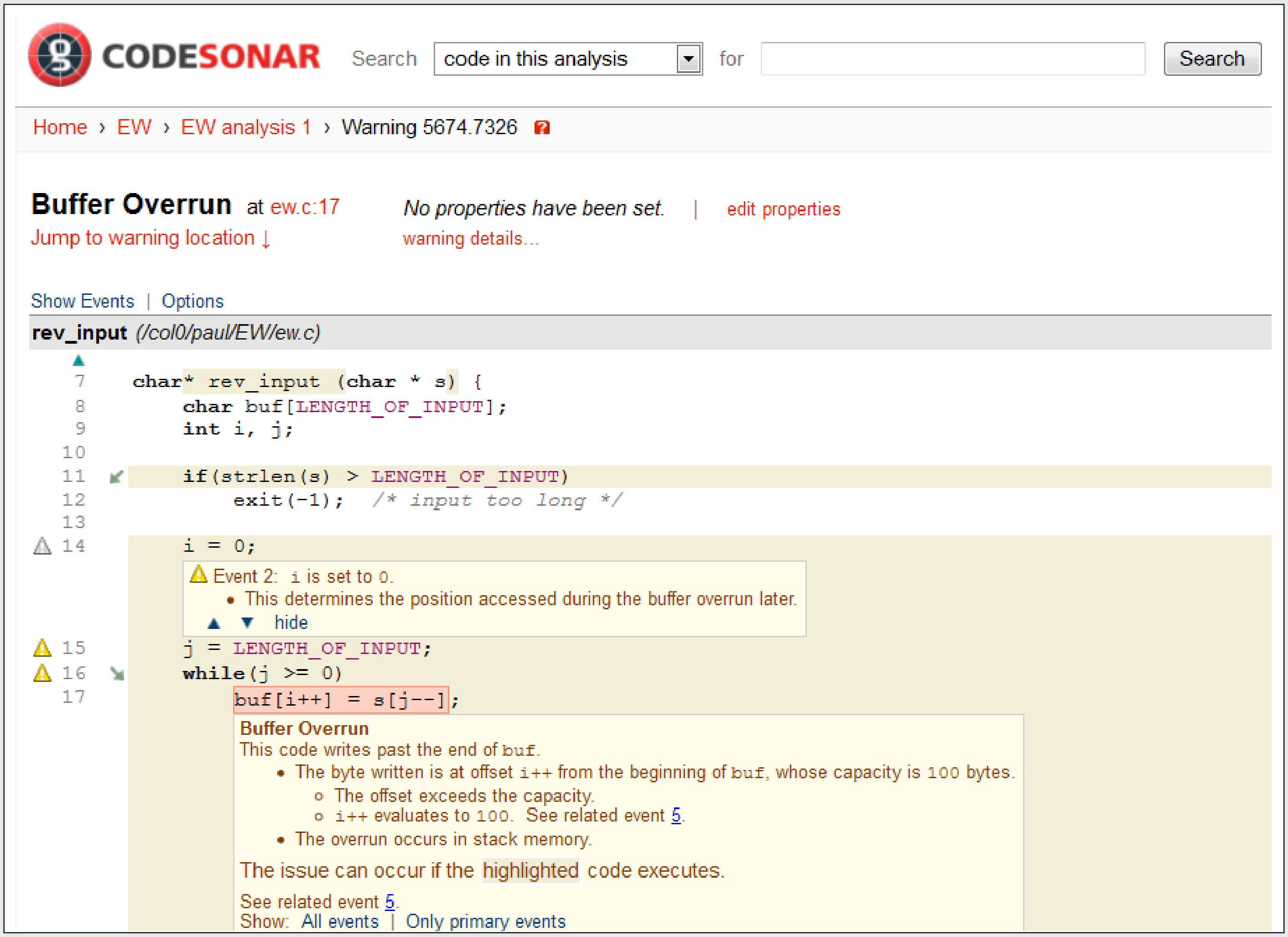
These may not cause the program to crash, but likely indicate that the programmer misunderstood an important property of the code. For example, a condition that is either always true or always false is unlikely to be intentional because it leads to dead code.

The really serious bugs – those that cause the program to fail, such as null pointer dereferences, buffer overruns, etc. – require sophisticated queries to find. Those queries can be thought of as abstract simulations. The analyzer simulates the execution of the program, but instead of using concrete values, it uses equations that model the abstract state of the program. If an anomaly is encountered, a warning is generated.

Static analysis tools are useful because they are good at finding defects that occur only in unusual circumstances, and because they can do so very early in the development process. They can yield value before the code is even ready to be tested. They are not intended to replace or supplant traditional testing techniques, but instead are complementary.

Figure 1 below shows an example buffer overrun warning report from CodeSonar. The report shows the path through the code that must be taken in order to trigger the bug. Interesting points along the way are highlighted. An explanation of what can go wrong is given at the point in which the overrun happens.

It can be difficult to track the flow of tainted data through a program because doing so involves tracking the value as it is copied from variable to variable, possibly across procedure boundaries and through several layers of indirection. Consider, for example, a program that reads a string from a risky network port. As strings in C are typically managed through pointers, the analysis must track both the contents of the string and the value of all pointers that might refer to the string. The characters themselves are said to be tainted, whereas the pointer is said to “point to taintedness.” If the contents of the string are copied, e.g., by using strcpy(), then the taintedness property will be transferred to the new string. If the pointers are copied, then the points-to-taint property must be transferred to the new pointer.



Home › EW › EW analysis 1 › Warning 5674.7326

Buffer Overrun at ew.c:17

Jump to warning location ↓

No properties have been set. | edit properties

warning details...

Show Events | Options

rev_input *(/col0/paul/EW/ew.c)*

```
 7   char* rev_input (char * s) {
 8       char buf[LENGTH_OF_INPUT];
 9       int i, j;
10
11       if(strlen(s) > LENGTH_OF_INPUT)
12           exit(-1);   /* input too long */
13
14       i = 0;
```

⚠ Event 2: `i` is set to `0`.

- This determines the position accessed during the buffer overrun later.

▲ ▼ hide

```
15       j = LENGTH_OF_INPUT;
16       while(j >= 0)
17           buf[i++] = s[j--];
```

Buffer Overrun

This code writes past the end of `buf`.

- The byte written is at offset `i++` from the beginning of `buf`, whose capacity is `100` bytes.
  - The offset exceeds the capacity.
  - `i++` evaluates to `100`. See related event 5.
- The overrun occurs in stack memory.

The issue can occur if the highlighted code executes.

See related event 5.

Show: All events | Only primary events

Figure 1. An example buffer overrun warning.

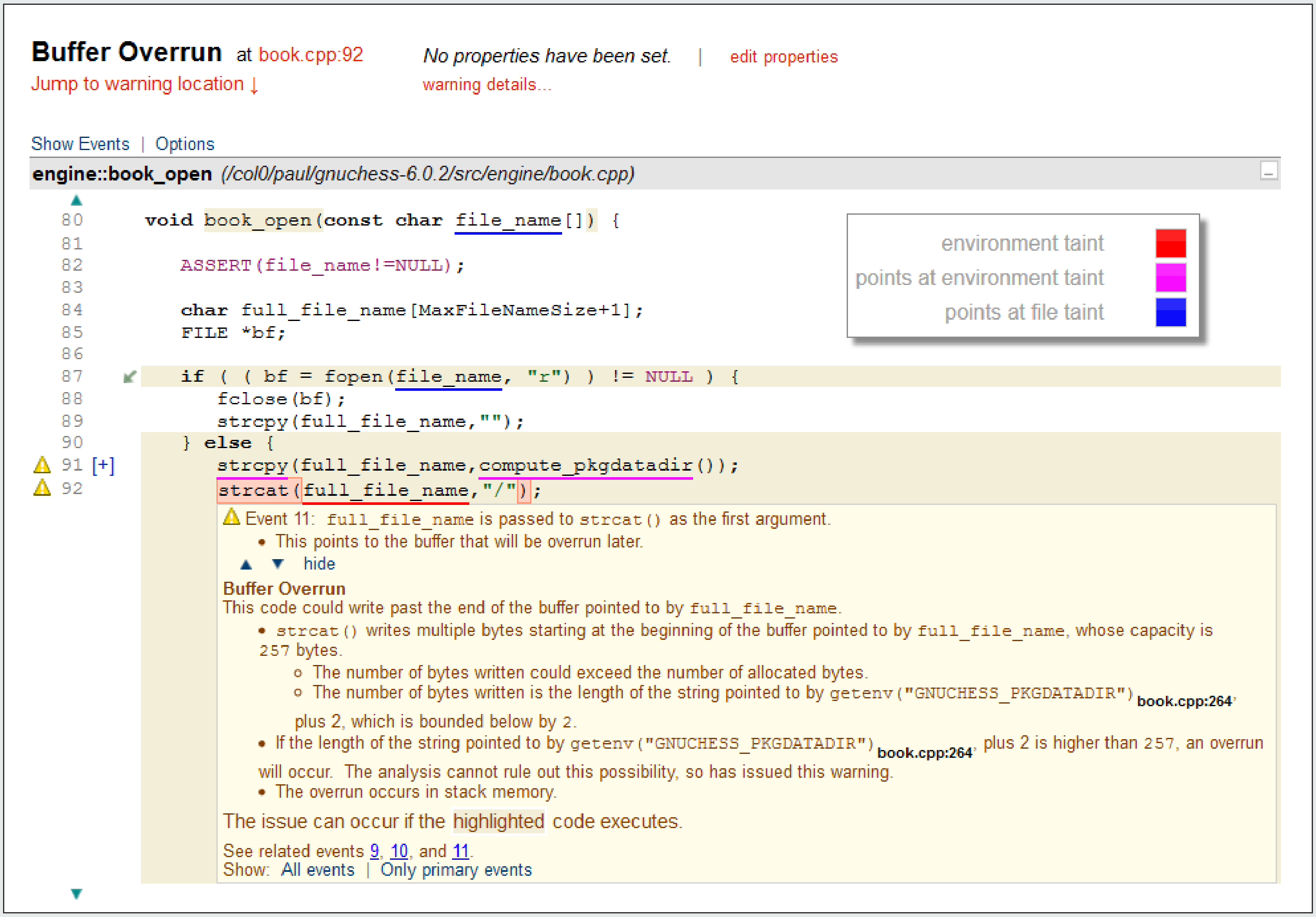
Of course, there may be pointers to those pointers, and even pointers to those, and the analysis must track those too. Ultimately the problem boils down to a kind of alias analysis, which is an analysis that can tell which variables access the same memory locations. An explanation of alias analysis is beyond the scope of this paper, but a good introductory article can be found here: www.wikipedia.org/wiki/Alias_analysis.

Understanding Taint Flow

Taint can flow through a program in unexpected ways, so an automated tool can also play an important role by helping programmers understand these channels. In CodeSonar, the location of taint sources and sinks can be visualized and program elements involved in flows can be overlaid on top of a regular code view. This can help developers understand the risk of their code and aid them in deciding how best to change the code to shut down the vulnerability.

Figure 2 below shows a report of another buffer overrun vulnerability.

In this example, first note the blue underlining on line 80. This indicates that the value of the variable pointed to by the parameter passed into the procedure is tainted by the file system. Although this fact may help a user understand the code, the most interesting parts of this warning are on lines 91 and 92. The underlining on line 91 indicates that the value returned by compute_pkgdatadir() is a pointer to some data that is tainted by the environment. The call to strcpy() then copies that data into the local buffer named full_file_name (declared on line 84). This, of course, transfers



Buffer Overrun at book.cpp:92
Jump to warning location ↓

No properties have been set. | edit properties
warning details...

Show Events | Options

engine::book_open *(/col0/paul/gnuchess-6.0.2/src/engine/book.cpp)*

```
   80     void book_open(const char file_name[]) {
   81
   82        ASSERT(file_name!=NULL);
   83
   84        char full_file_name[MaxFileNameSize+1];
   85        FILE *bf;
   86
   87        if ( ( bf = fopen(file_name, "r") ) != NULL ) {
   88           fclose(bf);
   89           strcpy(full_file_name,"");
   90        } else {
⚠ 91 [+]        strcpy(full_file_name,compute_pkgdatadir());
⚠ 92             strcat(full_file_name,"/");
```

environment taint	🟥
points at environment taint	🟪
points at file taint	🟦

⚠ Event 11: `full_file_name` is passed to `strcat()` as the first argument.

- This points to the buffer that will be overrun later.

▲ ▼ hide

Buffer Overrun
This code could write past the end of the buffer pointed to by `full_file_name`.

- `strcat()` writes multiple bytes starting at the beginning of the buffer pointed to by `full_file_name`, whose capacity is `257` bytes.
    - The number of bytes written could exceed the number of allocated bytes.
    - The number of bytes written is the length of the string pointed to by `getenv("GNUCHESS_PKGDATADIR")` book.cpp:264' plus 2, which is bounded below by `2`.
- If the length of the string pointed to by `getenv("GNUCHESS_PKGDATADIR")` book.cpp:264' plus 2 is higher than `257`, an overrun will occur. The analysis cannot rule out this possibility, so has issued this warning.
- The overrun occurs in stack memory.

The issue can occur if the highlighted code executes.

See related events 9, 10, and 11.
Show: All events | Only primary events

Figure 2. A buffer overrun warning where the underlining shows the effect of taint.

the taintedness property into that buffer. Consequently, on line 92, the red underlining shows that the buffer has become tainted by a value from the environment.

The explanation for the buffer overrun confirms that the value returned by compute_pkgdatadir() is in fact a value retrieved from a call to getenv(). A user inspecting this code can thus see that there is a risk of a security vulnerability if an attacker can control the value of the environment variable.

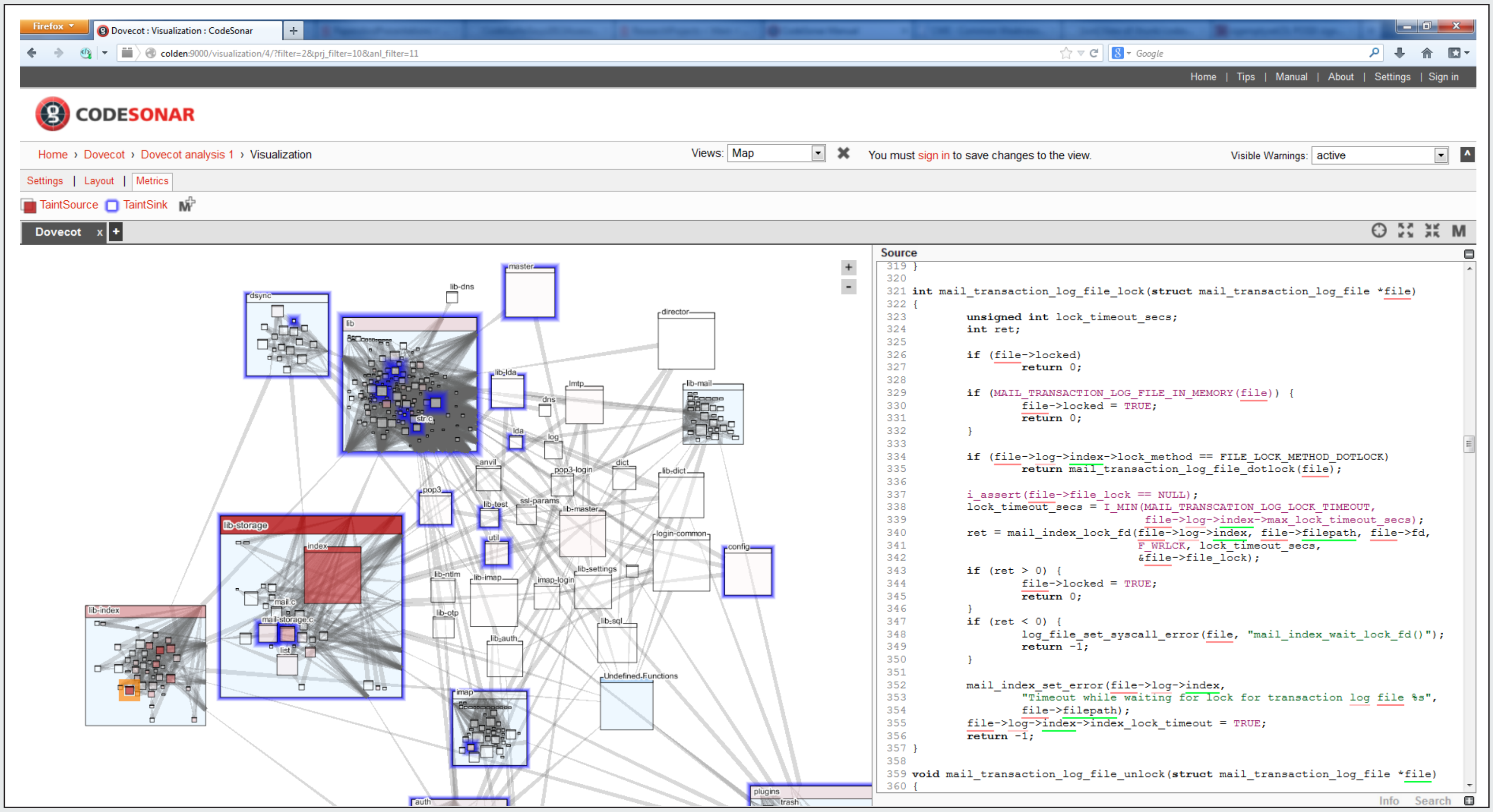
In CodeSonar, an alternative way of viewing the flow of taint through a program is a top-down view. An example is shown below in Figure 3.

In this example, the user has made use of the red coloration to identify a module containing taint sources. This is a reasonable approximation of the attack surface of the program. The code within that module is shown in the pane to the right; the underlining shows the variables that carry taint.

Conclusions

Software that assumes that its inputs are well-formed and within reasonable ranges is inherently risky and prone to failure. In the worst case, bad data can lead to serious security vulnerabilities and crashes.

Taint analysis is a technique that helps programmers understand how risky data can flow from one part of the program to another. An advanced static-analysis tool can run a taint analysis and present the results to the user, making the task of understanding a program's attack surface easier, and easing the work involved in finding and fixing serious defects. ■



```
}

int mail_transaction_log_file_lock(struct mail_transaction_log_file *file)
{
        unsigned int lock_timeout_secs;
        int ret;

        if (file->locked)
                return 0;

        if (MAIL_TRANSACTION_LOG_FILE_IN_MEMORY(file)) {
                file->locked = TRUE;
                return 0;
        }

        if (file->log->index->lock_method == FILE_LOCK_METHOD_DOTLOCK)
                return mail_transaction_log_file_dotlock(file);

        i_assert(file->file_lock == NULL);
        lock_timeout_secs = I_MIN(MAIL_TRANSCATION_LOG_LOCK_TIMEOUT,
                                  file->log->index->max_lock_timeout_secs);
        ret = mail_index_lock_fd(file->log->index, file->filepath, file->fd,
                                 F_WRLCK, lock_timeout_secs,
                                 &file->file_lock);
        if (ret > 0) {
                file->locked = TRUE;
                return 0;
        }
        if (ret < 0) {
                log_file_set_syscall_error(file, "mail_index_wait_lock_fd()");
                return -1;
        }

        mail_index_set_error(file->log->index,
                "Timeout while waiting for lock for transaction log file %s",
                file->filepath);
        file->log->index->index_lock_timeout = TRUE;
        return -1;
}

void mail_transaction_log_file_unlock(struct mail_transaction_log_file *file)
{
```

Figure 3. A top-down view of the call graph of the program showing modules according to the physical layout of code in files and directories. The red coloration shows the modules with the most taint sources, and the blue "glow" shows modules with taint sinks.

About GrammaTech

GrammaTech's tools are used by software developers worldwide, spanning a myriad of embedded software industries including avionics, government, medical, military, industrial control, and other applications where reliability and security are paramount. Originally spun out of Cornell's computer science labs, GrammaTech is now both a leading research center for software security and a commercial vendor of software-assurance tools and advanced cyber-security solutions. With both static and dynamic analysis tools that analyze source code as well as binary executables, GrammaTech continues to advance the science of superior software analysis, providing technology for developers to produce safer software. To learn more about GrammaTech, visit www.grammatech.com.

For more information:
www.grammatech.com
Email: info@grammatech.com

GrammaTech, Inc. Headquarters
531 Esty Street
Ithaca, NY 14850

U.S. sales: (888)695-2668
International Sales: +1-607-273-7340
Email: sales@grammatech.com

GRAMMATECH