CUTTER IT JOURNAL

Applying Aspect-Oriented Programming to Security

by John Viega, J.T. Bloch, and Pravir Chandra

Aspect-oriented programming (AOP) is a new programming paradigm that explicitly promotes separation of concerns. In the context of security, this would mean that the main program should not need to encode security information; instead, it should be moved into a separate, independent piece of code. The object-oriented paradigm can sometimes separate concerns in an intuitive manner by grouping concerns into objects. However, the object-oriented paradigm is only good at separating out concepts that map easily to concrete objects. It does not map well to abstract notions. For example, it is difficult to model security in the object-oriented paradigm. While one could write a central SecurityManager class for an application, calls to this class and important checks will necessarily be spread throughout the code base. Unfortunately, if a critical check is forgotten in an important spot in the code, the central class will not have the opportunity to recover. This problem exists because security is a concern that affects the entire system in a broad way. Aspect-oriented programming can solve this problem by allowing security concerns to be specified modularly and applied to the main program in a uniform way.

We have built an aspect-oriented extension to the C programming language.[1] While this extension is a general-purpose one, it has significant benefits in the realm of security. The aspect-oriented technique allows security policies to be separate from the code, enabling developers to write the main application and a security expert to specify security properties. Also, this technique can abstract away much of the expert knowledge currently required for writing secure code, allowing developers to reasonably secure their software even if a security expert is not available to assist.

Our approach is general enough to be applied to any language, and in the future we anticipate experimenting with extensions for several different languages. At present, we focus on C because there are several classes of common security vulnerabilities in that language. Further, despite any drawbacks of C, it is still widely used to write security-critical applications.

MOTIVATION

Experience has shown that developers aren't very good at writing secure software. Part of the problem is a lack of education; few classes cover this material, and no books cover it well. However, education isn't the entire issue, as evidenced by the fact that buffer overflow exploits in C code are quite common, even in software written by developers who know a lot about this problem.

We think it should be easier to design security into an application, instead of relying on the "penetrate-and-patch" approach to security, where problems are addressed in an ad hoc manner, generally as flaws are revealed in the field. Unfortunately, the penetrate-and-patch approach to security is widespread, as opposed to the alternative of designing software with security in mind.

[1]All work described in this article was done at Cigital, Inc.

Developers continue to write insecure code and to fix security problems when someone happens to notice them.

There are two primary reasons why this problem is so prevalent. The first is that no coherent design time methodologies or tools are widely available. In fact, there are no comprehensive resources available to help write secure programs; developers do not know what the problems are. As a result, developers continue to write insecure code and to fix security problems when someone happens to notice them. The second problem is that designing and implementing secure systems currently requires a lot of expert knowledge. Even with good methodologies, good tools are still critical to alleviating this problem, as the average developer is likely to be either unwilling or unable to use the methodology effectively if no tools are available.

Another large part of the problem is the state of programming languages from a security point of view. A few languages have given significant thought to security primitives that should be present to help programmers write better code. However, two of the most widely used programming languages, C and C++, present significant security risks. This is because many of their standard features can easily be used in such a way as to inadvertently leave a security flaw in a program. Even languages with significant security architectures, such as Java, leave something to be desired. While looking at many commercial products, we have found that a large percentage of applications have significant security problems that are present in the design phase and persist through to implementation despite the language used. Some of the more common problems include misuse of security protocols and an unrealistic view of what a system should consider "trusted." Languages have yet to make significant inroads in these areas.

PRINCIPLES

In talking with many developers about security, we have noticed that even security-aware people find it much easier to write their program and then later go back and try to "bolt on" security as an afterthought. In our experience, this approach doesn't work; Currently, security generally needs to be designed into an application from the beginning. There is a fundamental conflict here between what works well in practice and the way developers work, even when they are well educated on security matters.

All tools we have encountered to help prevent security vulnerabilities in general-purpose programming languages are after-the-fact tools, such as vulnerability analysis tools that look for common programming and configuration errors. They do not address how a developer should design and implement software in a security-conscious manner. Our aspect-oriented extension provides a more proactive approach to this problem. We designed the language with the following principles in mind:

- The amount of expert knowledge necessary to secure source code should be minimized. It should not be easy for a developer to accidentally introduce security problems into a program just because the language and the concepts of secure programming haven't been mastered. Common language pitfalls should be averted, and the programmer should be protected from common classes of mistakes that are not language specific.
- The security-related elements in a program should be abstracted out of the program proper, for the sake of clarity, maintainability, and reuse.
- Security policies should be defined using a language general enough that it is

possible to create new policies to deal with application-specific issues or previously unknown security vulnerabilities.

- An emphasis should be placed on "security by default," so that the level of effort for developing secure applications is minimized. This is a bigger challenge for C than for most other languages, since the language and standard libraries contain so many unsafe operations.
- Currently, if a security consideration is omitted in just *one* place, it can easily lead to a flaw. It should be easy to express policies about the program that apply generically to a consideration, and then have the policy be applied through the program automatically. For example, it should be possible to say that no secret information is ever sent over the network unencrypted (perhaps the actual implementation may choose to encrypt all data to satisfy this property).
- Legacy source code with known or potential security problems should be able to benefit from such a tool; the amount of new code necessary should be minimized. It should also be possible to avoid modifying the original program, since doing so tends to introduce errors.
- When it makes sense to do so, security policies should be reusable across different applications.

LANGUAGE

The purpose of our aspect language is to specify structured transformations on a program. Usually this involves inserting code at well-defined points. However, it may occasionally involve removing code. Once we have specified points of interest in our code, there are three things we can do:

1. Insert code before points of interest
2. Insert code after points of interest
3. Replace the code at the point of interest

There are several types of locations we can operate upon:

1. **Calls to functions.** For example, we can replace all calls to functions prone to buffer overflows with safe alternatives.
2. **Function definitions.** Inserting code around a function definition will ensure the code runs every time the function runs. Just operating on calls to functions will not catch calls from third-party libraries.
3. **Pieces of functions.** If a function uses jump labels, we can operate on a line of code following a label, or we can operate on a block of code between two labels.

In our language, an aspect is a programming construct that lives in its own file. It specifies points of interest and operations to be applied on those points of interest.

Here is an example aspect that allows us to replace all calls to the rand() call with a secure replacement. The rand() call is undesirable because its output is completely predictable and reproducible.

```
aspect secure_random {
  int secure_rand(void) {
    /**
     * Secure call to random defined here.
     */
  }

  funcCall<int rand(void)> {
    replace {
      secure_rand();
    }
  }
}
```

If a security consideration is omitted in just *one* place, it can easily lead to a flaw.

The function secure_rand is like any other C function. In this example, it will implement a secure random number generation algorithm. The “funcCall” keyword specifies that we wish to match calls to functions. The function we wish to match is specified between the angle brackets. In this case, we wish to match calls to rand. We specify the function by its signature, noting that rand takes no arguments and returns an integer. The “replace” keyword specifies that we wish to run code instead of calling rand(). The code inside the replace block will run instead.

At compile time, our language takes any aspects, along with the regular C program, and “weaves” them into a single C program, which is then compiled.

Wildcarding

Allowing the programmer to easily specify patterns describing interesting constructs makes the aspect language significantly more powerful, but it also introduces important subtleties into the language. We support three types of matching facilities: name, type, and argument matching.

Name matching allows the programmer to indicate an interest in functions, files, or modules whose names match a pattern. We use the ? (question mark) construct to specify wildcards in names. The wildcard construct stands for zero or more characters. For instance, foo? matches foo, foobar, and foofoofoo, but it would not match barfoo.

Using name matching, the programmer can focus on the functions int foo(char), and int foobar(char) by specifying the pattern int foo?(char). However, additional facilities are required to specify an interest in functions that take a single argument, where we don’t care what type of argument it is. To support such functionality, our language has an *any* keyword, which can be used in place of any type specifier. For example, we could say any ?(any) to match all functions taking one parameter, or any ?(any *) to specify all functions with one parameter that is a pointer.

In order to match variable argument functions, we support a “…” operator. For example, to match calls to the fprintf function, we can specify fprintf(FILE*, char *, ...).

Context Gathering

The code constituting a transformation will often depend on the site matching the aspect model. Our language provides variables containing useful contextual information. For instance, a transformation may need information about the function containing the matching site, or the module containing that function, or even the file currently undergoing transformation. Just as the preprocessor variable __FILE__ provides the fully qualified file name, the aspect language variables __FUNCTION__, __MODULE__, and __ASPECT__ provide information about the function, module, and aspect related to the current transformation. We expect this set of context-gathering variables to expand in future versions of the language.

Order and Precedence Concerns

The order in which transformations are applied and how conflicts between transformations are arbitrated represent an area remaining open to inquiry. To see that application order is an issue and conflicts are possible, consider transformations A and B, where A removes all calls to the insecure function exec(...) and B logs information about all such invocations. If the transformations are applied in order AB, we would not expect information about exec calls in the log file, since they are all removed. If transformations are applied in order BA, the log file would remain empty, but the logging code will never get executed. Which of these alternatives is desirable? We currently ignore this problem, disallowing all conflicts. One promising option is to provide guidance to the compile-time "weaver" by specifying aspects that govern the order in which other aspects transform the source code.

APPLYING AOP TO SECURITY

This general-purpose programming language has many applications to security. We previously gave an example of replacing insecure function calls with secure replacements, but there are plenty of other uses in this domain:

- It can be used to automatically perform error checking on security-critical calls.
- We can use it to implement the StackGuard technique of buffer overflow protection, inserting special code at function entry and exit.
- It can automatically log data that may be relevant to security.
- It can be used to replace generic socket code with SSL socket code.
- It can automatically insert code at startup that goes through a set of "lock-down" procedures that most programmers would not add to their programs.
- It can be used to specify privileged sections of a program and to automatically request and return privileges when appropriate.

An aspect weaver with a suite of aspects that address security concerns is generally language independent. In general, a security concern such as secure random generation is independent of the language in which the application is written. A small exception to this is the case in which a security concern does not exist with the language being used; for example, buffer overflows don't exist in applications written in pure Java. In cases such as this, the security concern can be thought of as being addressed by a null aspect — one with no transforms. In any case, we can assemble a list of our security concerns in some meta-language such as a simple configuration file that lists all the security concerns we have, along with any special language-independent parameters for them (e.g., specify "secure random numbers" as a security concern and specify "my-secure-algorithm" as a parameter to the aspect that will address this problem). Given a common interface for the aspects that address the same concerns regardless of the language the aspect was written for, we can create a configuration file that, given a weaver for the language in which our source is written and a suite of security aspects for that language, can be applied to any program source and assure it is secure against the concerns noted within. This meta-language provides a powerful tool for people to automate the securing of their applications.

Example

The following example shows a more practical example of how our technique applies to real programs. First, we build a library of security aspects that we wish to apply to our program. For example, we might wish to replace calls to rand and perform automatic error checking on calls to malloc (though this is only rarely a security problem and is more a reliability problem):

```
aspect secure_random {
  int secure_rand(void) {
    /**
     * Secure call to random defined here. Any other accompanying
     * functions can be defined in the same scope as this function.
     */
    return 0;
  }

  void support_function(void) {
  }

  funcCall<int rand(void)> {
    replace {
      secure_rand();
    }
  }
}

aspect malloc_check {
  funcCall<void * malloc(size_t x)> {
    after {
      if(__RETVAL__ == NULL) {
        printf("malloc(%i) failed in function %s(). exiting...\n",
               x, __FUNCTION__);
        exit(-1);
      }
    }
  }
}
```

Now, we take the C program to which we want to apply our aspects:

```
int init_msg(char *[], int);

int main(void) {
  char * msg[4];
  int i;

  init_msg(msg, 4);
  i = rand() % 4;
  printf("%s",msg[i]);
  return 0;
}


int init_msg(char * arg[], int size) {
  int i;
  for(i=0 ; i<size ; ++i) {
    arg[i] = (char *) malloc(20);
    sprintf(arg[i], "Message Number %d\n", i);
  }
  return 0;
}
```

We may wish to add our own aspects for debugging purposes at this point:

```
aspect debug {
  /* Notify when init_msg starts and stops. */
  funcDef<void init_msg(any)> {
    before {
      printf("Entering init_msg.\n");
    }
    after {
      printf("Leaving init_msg.\n");
    }
  }
}
```

Our compiler takes all of the above code and conceptually generates the following C program:

```
int secure_rand(void);
void support_function(void);
int rand_wrap(void);
void * malloc_wrap(char *, size_t);
int init_msg(char *[], int);

int main(void) {
  char * msg[4];
  int i;

  init_msg(msg, 4);
  i = rand_wrap() % 4;
  printf("%s",msg[i]);
  return 0;
}

int init_msg(char * arg[], int size) {
  int i;

  printf("Entering init_msg.\n");
  for(i=0 ; i<size ; ++i) {
    arg[i] = (char *) malloc_wrap("init_msg", 20);
    sprintf(arg[i], "Message Number %d\n", i);
  }
  printf("Leaving init_msg.\n");
  return 0;
}

int secure_rand(void) {
  return 0;
}

void support_function(void) {
}

int rand_wrap(void) {
  int retval;
```

The space of available software security assurance is currently inhabited primarily by small, open-source tools that address only a fraction of the actual problem.

```
  retval = secure_rand();
  return retval;
}

void * malloc_wrap(char * FILENAME, size_t arg1) {
  void * retval;
  retval = malloc(arg1);

  if(retval == NULL) {
    printf("malloc(%i) failed in function %s(). exiting...\n",
           arg1, FILENAME);
    exit(-1);
  }
  return retval;
}
```

RELATED WORK

The aspect-oriented programming paradigm was first introduced by Gregor Kiczales [7]. Xerox PARC developed the first aspect-oriented programming language, AspectJ,[1] which is an extension to the Java programming language.

The space of available software security assurance is currently inhabited primarily by small, open-source tools that address only a fraction of the actual problem. There are multiple patches for the gcc compiler that implement array bounds checking. There are several tools that provide some sort of security against buffer overflow attacks, including StackGuard [2] and FIST [5]. However, most of these tools are solely interested in buffer overflows.

Another type of tool in the security assurance domain is the "secure data flow" tool. Examples of this tool are the "taint" version of Perl and the JFlow programming language (a Java extension) [8]. In such tools, data is labeled either "untrusted" or "trusted." "Untrusted" data cannot be passed to trusted items without the programmer explicitly allowing it. Similarly, "trusted" data cannot be passed to "untrusted" items for fear of leaking secret information (unless it has been explicitly declassified by the programmer).

All of the capabilities of the above tools could be described in a security specification and woven into a program using our aspect-oriented security approach. We hope to incorporate existing tools as off-the-shelf technology whenever possible. For example, the aforementioned tools for preventing buffer overflows can potentially be leveraged in the implementation of our approach. We hope to provide a uniform and general-purpose interface to these tools, while adding a large amount of flexibility and extensibility; only a few of the many problems we seek to handle are addressed by current tools.

Commercial tools in the security assurance space are almost universally general purpose and not security specific. For example, there are many tools such as Rational's Purify that can help find and fix buffer overflow problems, even though the tool is not specifically a security tool [6].

Another class of tools is the after-the-fact tools that support the penetrate-and-patch model. These tools generally are concerned with taking preexisting source code and identifying potentially dangerous constructs based on a database and some static analysis. Currently, the only publicly available tool for source code analysis is ITS4, which scans C and C++ code for over 100

[1]See www.aspectj.org.

potential problems [10]. Wagner has a buffer-overflow scanner that performs a more sophisticated analysis; however, it is not publicly available and is limited in scope [11]. There are similar general-purpose tools that may catch some security bugs, including lint tools such as LCLint [3].

Previous work has also been done in policy languages for security. Most such languages specify file access control, allowing the programmer to give explicit policies stating what a program can and cannot do to files. Examples of such systems include Naccio [4], Ariel [9], and PolicyMaker [1]. We anticipate incorporating this sort of tool as a small part of our total functionality.

SUMMARY

We have identified some of the major problems plaguing software security and discussed how separating security from a program itself might help alleviate these problems. We have built an extension to the C programming language for defining security concerns and implementing a weaver that generates and integrates code into C programs.

REFERENCES

1. Blaze, M., J. Feigenbaum, and J. Lacy. “Decentralized Trust Management.” In *Proceedings of the 17th IEEE Symposium on Security and Privacy*. IEEE, 1996.

2. Cowan, C., et. al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.” In *Proceedings of the Seventh USENIX Security Symposium*. USENIX Association, 1998.

3. Evans, D., J. Guttag, J. Horning, and Y. Meng Tan. “LCLint: A Tool for Using Specifications to Check Code.” In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 1994.

4. Evans, D., and A. Twyman. “Flexible Policy-Directed Code Safety.” In *Proceedings of the 1999 IEEE Symposium on Security and Privacy.* IEEE, 1999.

5. Ghosh, A., T. O’Connor, and G. McGraw. “An Automated Approach for Identifying Potential Vulnerabilities in Software.” In *Proceedings of the 1998 IEEE Symposium on Security and Privacy.* IEEE, 1988.

6. Hastings, R., and B. Joyce. “Purify: Fast Detection of Memory Leaks and Access Errors.” In *Proceedings of the Winter USENIX Conference.* USENIX Association, 1992.

7. Kiczales, G., et al. “Aspect-Oriented Programming.” In Proceedings of the European Conference on Object-Oriented Programming (ECOOP) ‘97. Springer-Verlag, 1997.

8. Myers, A. “Practical Mostly-Static Information Flow Control.” In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 1999.

9. Pandey, R., and B. Hashii. “Providing Fine-Grained Access Control for Mobile Programs Through Binary Editing,” Technical Report CSE-98-8. University of California, Davis, 1998.

10. Viega, J., J.T. Bloch, T. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC 2000)*. IEEE, 2000.

11. Wagner, D., J. Foster, E. Brewer, and A. Aiken. “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.” In *Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS)*. Internet Society (ISOC), 2000.

John Viega is a senior developer and researcher at Widevine Technologies. He is the coauthor of two upcoming books, Building Secure Software *(Addison-Wesley) and* Java Enterprise Architecture *(O’Reilly). He also writes a regular column on software security for IBM’s* developerWorks.

J.T. Bloch is a developer at Widevine Technologies and a researcher in computer security. He is one of the developers of the ITS4 tool for auditing C and C++ source code. He is currently working on software security issues in untrusted environments.

Pravir Chandra is a developer at Widevine Technologies and a researcher in computer security. He is currently working on software security issues in untrusted environments.

The authors can be reached at Widevine Technologies, 11951 Freedom Sq., Suite 1333, Reston, VA 20190, USA. Tel: +1 206 254 3000; Fax: +1 206 254 3000; E-mail: johnv@widevine.com, jt@widevine.com, and pchandra@widevine.com.