Runtime Defenses

Nicholas Carlini

7 September, 2011

1 Baggy Bounds Checking

1.1 Motivation

There are large amounts of legacy C programs which can not be rewritten entirely. Since memory must be manually managed in C, and bounds are not checked automatically on pointer dereferences, C often contains many memory-saftey bugs. One common type of bug is buffer overflows, where the length of an array is not checked before copying data in to the array: data is written past the end of the array and on top of whatever happens to be at that memory location.

1.2 Baggy Bounds Checking

Baggy bounds checking was proposed as a possible defense which would detect when pointers wrote to a memory location out of bounds. It is a runtime defense, which uses source code instrumentation. At a very high level, baggy bounds checking keeps track of bounds information on allocated objects. When a pointer which indexes in to allocated memory is dereferenced, bounds checking code ensures that the pointer still is indexing the same allocated memory location. If the pointer points out of bounds of that object, the program is halted.

Baggy bounds checking uses a buddy allocator to return objects of size 2^n, and fills the remaining unused bytes with 0s. A bounds table is created containing the log of the sizes of every allocated object. Using this table allows for single-memory-access lookups, which is much more efficient than other methods which require multiple memory accesses.

1.3 Baggy Bounds Checking Failures

Baggy bounds checking does not work in all situations, however. Consider the C program below.

```
struct User {
  char username[128];
  int is_authenticated;
  (void * fnptr)();
}
```

Assume the C program contains an overflow in the `username` buffer of the struct.

Since baggy bounds checking will only detect problems when a pointer is dereferenced in a different allocated object, the `username` buffer could still be overflowed to overwrite either the authenticated flag or the function pointer. This gives no benefit over typical uninstrumented code.

One disadvantage of this is it means that any program which writes it's own allocator in order to take advantage of specific program-specific properties (which may make the allocator able to do a better job than the normal `malloc`) will not be able to make use of baggy bounds checking.

1.4 Comparison with Java

It would be good to know how the speed of an instrumented C program compared to a Java program in terms of speed: if the Java program was just as fast, then it would be much safer to just use Java for new projects. However, for legacy programs, it is not always possible to rewrite the entire project in Java, even if it turns out that Java was just as fast.

1.5 Compile Time Bounds Checking

This brings rise to a question: why doesn't C do what Java does and put compile-time check bounds on the buffers? This works in some cases, and related work has done things like this when the size of the buffer is known statically. However, arrays can be created dynamically where the compiler will not know the bounds.

For example, consider the line of C:

```
char x = p->buf[i];
```

This case can be checked statically by inserting the bounds check, assuming the program knows that `p` is of type `struct User *`.

```
if (i >= 0 && i < 128) {
  exit(1);    // Oh no!
}
char x = p->buf[i];
```

However, the more difficult case is when a function takes a `void*` pointer as an argument, with no idea who passed the pointer or what information there is about the pointer. In order to solve this, we would need a bounds table to look up this information.

1.5.1 Uninstrumented Code

It is often the case in large programs that not all of the source code is available. It may be that a binary exists for a library, but there is no source code to recompile that binary under the new compiler. In this case, this pre-existing source code is called "uninstrumented code".

In the case of uninstrumented code, baggy bounds checking does absolutely nothing. All allocations done in the uninstrumented code are allowed to write to anywhere in memory, and no pointer dereferences inside of uninstrumented code is checked. While the latter case is obvious, the former case is a major drawback of baggy bounds checking: if an uninstrumented library returns any buffer, all memory accesses to that buffer are unchecked.

1.6 Instrumenting Binaries

This leads to a question of why bounds checking is not done on binaries. The reason for this is almost exclusively that instrumenting binaries causes a significant speed decrease — typically, binary-instrumented code is five to ten times slower than the original program. This is for two main reasons:

- *Static analysis is more difficult.* When dealing with the source code, it is much easier to determine the control flow graphs, and infer from there what checks are redundant. Instead of bounds checking in a loop that touches every item of an array once, the bounds can be checked only at the beginning.
- *Finding registers to do intermediate calculations is difficult.* During the register allocation phase, compilers try very hard to use all the registers available to them at every moment. When instrumenting the binaries, in order to do even a simple lookup sequence with only a single memory load, it might be required to temporarily write two or three registers to memory in order to have space to do an intermediate calculation.

One such tool which instruments binaries is called Valgrind. It keeps a 512mb where each bit represents whether the corresponding byte in memory is valid at that moment. That is, if `a` is an address then `T[a] == 1` iff the memory location `a` is readable and writable. To do this checking, it emulates the X86 instructions and finds any memory accesses and checks if the pointers are valid before doing the reads or writes.

This does not even try to detect when an object writes to invalid-but-allocated memory; if it tried to do that then the program would only get slower.

1.7 Fat Pointers

Fat pointers were a strategy originally proposed to keep track of bounds information along with pointers. A pointer is "fat" in that it contains the base of the array and the length of the array along with the pointer. When the program increments the pointer, only the pointer portion is incremented: the base and length are not. Then, when the pointer is dereferenced, the program can check that the pointer is within the bounds of the array by verifying it is larger than the base and smaller than the base plus the length.

This has a two main advantages over the baggy bounds checking. First, the bounds table is not required. This means the memory will only be reserved as it is needed on a pointer by pointer basis. Second, the bounds of the pointer are stored on the same page as the pointer itself, so it is unlikely that the program will pagefault when looking up the bounds of the array.

1.7.1 Issues

Despite the advantages, there are many issues with fat pointers. One of the major issues is that most C programs assume pointers on a 32 bit system will be only four bytes — a programmer may write `malloc(4*n)` instead of `malloc(sizeof(void*)*n)`, and in this case, a fat pointer will not have space to be placed.

The problems because much more difficult when dealing with uninstrumented libraries. These libraries will expect only pointers, and not the fat pointers. It is not too difficult to modify pointers when passing them as single arguments to functions in uninstrumented libraries: instead of passing the fat pointer, only the pointer portion can be passed. This does mean that uninstrumented code will benefit from the fat pointers, but it will at least work correctly.

While the previous case was possible, it is significantly more difficult to deal with structs of pointers. When passing a struct, which may contains pointers, between instrumented code and uninstrumented code, the uninstrumented code would not know about the fat pointers. One option to solve this would be to maintain separate objects — one with fat pointers, and one with normal pointers — and reflect all changes on the fat pointers object in the other and vice versa. However, this leads to the problem that if one program passes the address of one of the memory locations to the uninstrumented code, it will see this as the address of the fat pointer structure and not its copy of the structure. In general, this is not an easy problem to solve.

This is the driving cause of bounds tables: bounds tables allow programs to work reasonably well with uninstrumented code, while fat pointers do not.

1.8 Use in Practice

Despite all this work that lead to baggy bounds checking, it is almost never used in practice, and this is due to many reasons.

- *C programs are designed for speed.* Adding baggy bounds checking slows down C programs by different amounts: while some slow down only 8% others slow down by a factor of two.
- *Baggy bounds checking restricts memory available.* Pointers are made invalid by making half of the address space be invalid, and a further block is used for the bounds table.
- *Programs must be recompiled with a new memory allocator.* Any fragile code which happens to work using the default memory allocator may not work with a different allocator.
- *Baggy bounds checking competes against ASLR+DEP.* These two defenses provide a good enough security assurance, and do not slow program execution.

2 Command Injection Vulnerabilities

2.1 Background

Command injection vulnerabilities have a long history. Two interesting examples of command injection vulnerabilities follow.

2.1.1 Captain Crunch

Command injection vulnerabilities go back to at least 1970 when "Captain Crunch" demonstrated that the using the whistle packaged in the Captain Crunch serial boxes, it was possible to place free phone calls. The whistle in these boxes happened to make a tone at 2600 Hz, which, as it turns out, this was the same frequency used by AT&T's long distance lines to send control information.

A person could use this whistle to place free calls to any phone number by posing as the trusted phone system. This would be done first by sending a tone to indicate the current call was over. Then, the user could send additional tones to indicate that a long distance call should be placed. By calling a long-distance toll-free number and then rerouting the call to a non-toll-free number after the call had gone through, free phone calls could be placed.

AT&T fixed this problem by separating the voice and control channel — that is, by moving the control channel out of band.

2.1.2 Dial-Up Modem

Many types of dial-up modem also contained command-injection vulnerabilities. In order for the computer to send commands to the modem instead of the internet, a specific three character sequence (+++) was used to indicate that the following message was intended for the modem and should not be relayed over the phone.

The first company to implement this designed the system such that the three plus characters would be followed by a short 100ms pause, and then followed by the sequence of commands to be executed. In order to avoid patten infringement, other companies replicated this but did not require a pause after the three plusses.

This means that if a user were to receive a malicious email that contained three plusses followed by some modem instructions, and the user happened to reply to it, when the modem received the three plusses it would interpret that as a modem command and would then begin to do whatever followed.[1]

[1] In fact, the company which patented the original design sent out press releases which contained three plusses, and at least one employee's email signature contained three plusses.

2.2 Modern Command Injection

Modern command injection is very similar to the previously described examples. Consider the following program:

```
system("/usr/bin/mail " + addr);
```

If the `addr` variable is not checked for invalid characters, then an address of `"foo; halt;"` will cause the computer to be shut down (assuming the user has the permission to do so).

Another example would be the following statement:

```
String statement = "SELECT * FROM users WHERE user=" + username;
```

A user who could make his name `John OR 1 == 1` could make the `SELECT` statement show a list of all users. Even worse, however, is the user could make his name `John; DROP TABLE users;`.

This is occasionally "fixed" by placing quotes around the username:

```
String statement = "SELECT * FROM users WHERE user='" + username + "'";
```

However this can be defeated by making a name of `John'; DROP TABLE users; --`, since the single quote will end the quoted text and the `--` will cause the rest of the line to be a comment.

2.3 Taint

Taint analysis tracks where input comes from. If input comes from the user, we treat it as untrusted. If the input comes from the program (e.g., as a static string or as data from a trusted server) then it is treated as trusted.

The paper has a detailed discussion of how taint should flow from place to place in an efficient manner.

2.4 Policies

Taint by itself has very little use. Where the taint becomes useful is in creating policies which govern what sources are allowed to flow to specific sinks. This paper does not spend much time discussing policies, even though policies are one of the critical aspects of using taint.

The one policy they give as an example is for SQL. SQL injection, as described earlier, is a very common problem on databases. It is possible to use taint to try to solve this issue. The specific policy that the paper proposed was to check SQL commands for $(OR)^t \mid (SELECT)^t \mid (FROM)^t \mid ...$ where x^t means that x tainted. That is, this policy checks to see if a SQL query contains any of those words in tainted locations, and if so, then assume that the query is dangerous.

2.4.1 Policy Failures

Even if a program can track taint correctly all of the time, writing correct policies is very difficult. The previous approach was an attempted black-list all words which are possibly dangerous. Blacklists are almost always incorrect, as this one demonstrates.

For example, the above policy did not block the use of single quotes, or semi-colons, both of which can be used in a query to do damage.

Not only is it exceedingly difficult to identify all words and characters which are dangerous according to the specification, many databases contain database-specific queries which are not in the specification — meaning a blacklist which worked on one server might not work on another server.

2.4.2 Other SQL Policies

Writing policies is really hard. There has been more recent work on writing better policies.

One such method is to parse the SQL query and then make sure that there are no user-written control characters. This method is much more secure — any user input that is treated as part of the SQL query syntax is clearly bad.

This is not as easy as it sounds: it is a nontrivial task to write a program which will parse the input the same way that the database will. In some cases, a database may parse an input not according to the specification, while another database may correctly parse the input according the specification; this means the validation program must let no malicious input past it, even if the database parses the query incorrectly.

Another issue that is exploited in many SQL injection attacks is that what the analysis program sees is not always the same as what the database sees. For example, assume that the verifier doing the taint analysis takes input UTF-16 encoded, but the database uses only plain ASCII. If an attacker wanted to enter a singlequote in the database, he would find a UTF-16 character which contained 0x27 as one of the bytes. The verifier would see this byte and the bytes around it as a single unicode character, and allow it, whereas when it reached the database, it would interpreted as multiple ascii, treating the 0x27 byte as a single quote.