Hello there!

Thank you for your interest in Ruth’s musings in security. I aim to read one academic paper in cryptography (theory) and computer security (application) each week. Here are the musings from the week 30/7/2016 to 5/8/2016. I am very far from being an expert on these topics, therefore, if you need to contact me, to report errors in my stuff, my email address is thisemailisnottruthless@gmail.com. Also, for clarification, this is not research that was done this week. This is research that was read by Ruth this week =P

This Week in Crypto: "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities"

by DAVID WAGNER, JEFFERY S. FOSTER, ERIC A. BREWER, ALEXANDER AIKEN

Premise: Buffer overflow vulnerabilities are a serious threat to security. According to various studies done, they account for between 23% to 50% of vulnerabilities. Buffer overflow is common because C is inherently unsafe. Array and pointer references are not automatically bounds-checked, so it is up to the programmer to do the checks themselves. While there are libraries written to improve upon existing functions by incorporating bounds checking into them, there are a variety of reasons why these have not been fully adopted:

- Functions such as strcpy() are familiar to programmers, who will use them out of habit.

- Functions in this library behave dissimilarly, making it challenging to remember and adopt. For example, the strn*() calls have arguments in different orders, and strncpy()/strncat() differ in whether they null terminate strings.

- Functions like strncpy() or strncat() encourage off-by-one bugs by having the “length” of a string not include the null terminator (i.e. string length is strlen(x) - 1)

- The most general and intuitive primitive snprintf() is not available on all systems, so portable programs cannot rely on it.

To avoid the tedious and possibly erroneous manual code review required to avoid these errors, past work have attempted various methods to scan large libraries for buffer overflow errors. For example, there is the runtime code-testing option of fault injection. However, this misses a large number of errors and can only be done at runtime, not prior to execution of the whole code. The authors propose a static analysis method, which is one that will parse the code and flag as many probable errors as possible. They also try to balance the scalability of the system with the precision of the error locating.

Sketch of Methodology: The authors do this static analysis by modeling memory as a constraint problem. Specifically, they take the source, parse it to generate integer constraint problems, then use a constraint solver to flag potential buffer overflows. It should be noted that the author’s main objective was to find errors that had evaded past code reviews (both manual and automated). Thus, this does not seek to find all possible errors, but simply to have a higher and more accurate detection rate than past methods. To overview the mathematics behind this would be too verbose for an article like this (even though it is very beautiful). Therefore, I will simply sketch the idea here.

Then the authors establish the mathematical model that they wish to use, considering integer ranges as elements in a partial order (i.e. with ⊆), then define the closure of a set S as [inf S, sup S]. They extend operators +, −, × to sets in a natural way (e.g. $A \times B = \{a \times b \mid a \in A, b \in B\}$). When the result of an operation is not a range, we take the range closure of it. This allows for commutativity, additive identities and multiplication by 0.
to act as expected (but not distributivity or additive inverses). Then, we can establish a system of constraints using the partial order (e.g. \( x + y \subseteq z, s - r \subseteq p \ldots \)). We define an assignment as a function, \( \alpha \) mapping ranges to the variables such that the system of constraints are satisfied. The least solution is the assignment that minimizes all ranges assigned. It can be shown that every constraint system has a unique least solution.

To begin, the authors model C strings as an abstract data type. What this means is that they are abstracted from the code into objects, which can then be copied, written or removed. Then, buffers are pairs of integer ranges, modeled with variables \((v)\) with an allocated size \((\text{alloc}(v))\) and a length \((\text{len}(v))\) that is currently in use. Using the BANE toolkit, each integer program variable \(v\) is associated with a range variable \(v\). Then, constraints are generated with this, for example, if \(v = e\) is a line in the code, then we generate constraint \(e \subseteq v\). For simplicity, the analysis is flow-insensitive, ignores functions that are difficult to model (e.g. \text{strcat}, double indexed arrays,union types). Also, model other function calls monomorphically, meaning that we merge information for all calls of a function \(f\), with one variable determining the range of the return value. Finally, we model structs by treating all structure objects with the same type as potentially aliased, and so use a single variable. Fields of the struct also are assigned a variable.

At this point, the authors can do a once-through search for unsafe properties of each \text{alloc}(v)\) and \text{len}(v)\) to check for buffer overflows. After this once-through, we solve the system of integer range constraints for a least solution, and use that to infer further vulnerabilities. This is done by considering each constraint in the form \(f(v_i) \subset (v_j)\), where \(f\) is an affine function on \(\mathbb{Z}^n\) (simplifying some arithmetic expressions further, such as max/min). Now form a directed graph where the \(v_i\) form vertices and a directed edge \(v_i \rightarrow v_j\) when \(f(v_i) \subset (v_j)\).The objective is then similar to a network flow algorithm, where information is propagated along the edges until all edges have \(f(\alpha(v_i)) \subset \alpha(v_j)\) for least assignment \(\alpha\), and for each edge as above. Note that in the case of cycles in the graph, we directly solve the constraint subsystem of each cycle before solving the acyclic subgraphs. This can be done in linear time.

**Experimental Results:** The authors prototype was applied to a dozen software packages, and they present the interesting results found:

- **Linux nettools package** (7k lines of code) was audited by hand in 1996, after buffer overflow vulnerabilities were found. This is the package that provides the source for functions like \text{route} or \text{ifconfig}. The authors found two new vulnerabilities that are remotely exploitable.

- **sendmail 8.9.3** (32k lines of code) was also audited by hand. A new off-by-one error was found by the authors, which was obscured from manual auditor's view by a complex calling pattern.

- **sendmail 8.7.5** This is an old version, so that the authors could check the errors found against bug reports. It managed to find many vulnerabilities, crediting the prototype with being useful towards real systems.

**Review of Method:** The method has good performance and scalability, due to the trade-offs made by the authors between precision and scalability. The 32k lines of sendmail took around 15 minutes to run on a fast Pentium III workstation.

The method has its limitations. Chiefly, this is the large number of false alarms due to the above trade-off. In sendmail 8.9.3, 44 errors were flagged, while only 4 were genuine errors. This suggests the need for more precision (and a slower running program), that cuts down on the time required to manually sieve through the false alarms. The authors propose moving this forward with the addition of other analysis techniques, such as Pratt's method or Integer Programming (Simplex Method).

---

**This Week in Security: "Practical Techniques For Searches on Encrypted Data"**

By **Dawn Xiaodong Song, David Wagner, Adrian Perrig**

**Premise:** The authors consider the case where Alice wishes to perform a word search on documents on an untrusted server, Bob. This could, for example, be a situation where a user is doing a word search on his emails. However, Alice would very much like to encrypt the documents before giving them to Bob, while maintaining Bob's ability to perform word searches for her. At the same time, a trivial solution such as using a deterministic encryption on the words would open the plaintext up to statistical analysis.

Therefore, the solution the authors seek is one that allows provable security as well as efficient word search results. In particular, the authors want to support hidden queries, so the user may ask the untrusted server to search for a secret word without revealing the word, as well as query isolation, meaning the untrusted server learns nothing more than the search result about the plaintext. In addition, we seek a scheme that will incur only \(O(n)\) overhead in block and stream cipher operations, to search a space of \(n\) words. Also, this should be easily extended to support more advanced searches (e.g. wildcards or phrases).

We formalize the situation by considering a set of documents that have been split into “words” of equal length (padding shorter words and breaking up larger words).
Alice will submit $W$, a word, to Bob, who will return to Alice the set of documents which contain this word.

The authors share their solution by first describing three apparent solutions that fail for varying reasons.

**Basic Scheme:** The first proposed scheme involves Alice encrypting her set of words $W_1 \ldots W_l$ to ciphertext $C_1 \ldots C_l$ such that $C_i = W_i \oplus T_i$ (bitwise XOR), and $T_i =< S_i, F_{k_i}(S_i) >$ where $S_i$ are pseudorandom bits generated by a stream cipher. Here, $F$ is a keyed pseudorandom function, and the $k_i$ are keys known to Alice. Then, when Alice wishes to search for $W_i$, he provides Bob $W$ and $k_i$ for each position $i$ that the word may occur. Bob then searches if any $C_i \oplus W$ take the form $< S, F_{k_i}(S) >$, for any $S$ and any of the $k_i$ provided, then Bob can return that document to Alice since, with high probability, it contains the word that Alice searches for.

At the positions where Bob does not know $k_i$, he gains no information on the words. However, the scheme only provides limited forms of privacy since she must reveal all the $k_i$ for all the segments that she wants to be searched, potentially revealing information about large sections of plaintext.

**Controlled Search** This is the same as the above method, but involves using a keyed pseudorandom function $f$ to derive the key from the word. Thus, $f(W_i) = k_i$. This way, by just revealing the word to Bob would allow Bob to search for the word, but then glean no information about any word other than the one that is being searched for. The only remaining problem now is that the word $W$ now has to be revealed to Bob.

**Hidden Searches:** A natural extension would then be for Alice to use a deterministic encryption, $E$ to pre-encrypt each word before beginning the above protocol.

While this seems to solve the problem above, this creates a more serious problem. Note that to calculate $k_i$, Alice would need $E(W_i)$, which she has stored on the server and does not have access to. She needs $k_i$ to calculate $F_{k_i}(S_i)$. Thus, the entire scheme is useless because Alice can no longer decrypt the information after she receives it.

**The Final Scheme:** Thus, it is imperative that $k_i$ be derived from bits that are not the bits output by function $F$. Thus, the message, post encryption, should be split into two sections (i.e. $E(W_i) =< L_i, R_i >$), where $k_i$ is generated from $L_i$. Note that $R_i$ should be the same length as the output of function $F$. Note also that the pre-encryption should resolve the propensity for different words to have the same prefix, eliminating the probability of false positives, up to the birthday bound.

The authors show that this scheme is provably secure, and that even when a single $k_i$ is revealed, no extra information is leaked beyond the ability to identify the positions where the corresponding word $W_i$ occurs.

**Other Concerns:** The authors also list a few additional notes, to extend the scheme's usability.

- There is a natural extension to supporting more advanced search queries, such as wildcards or multiple words. However, this can lead to dramatically more information leakage (to a potential statistical attacker).
- In addition, the attacker’s knowledge of the frequency of the searched word cannot be overlooked, since he may be able to glean information from the fact that certain documents have more of a particular word than others.
- The authors propose a simple solution that directly addresses the second and is related to the first of these issues. This is to append to the plaintext of each word a counter noting the number of times that word has been entered into the document, so that the plaintext for each of these identical words are actually different. This allows one to search the existence of a word with $< W, 0 >$, or if a word occurs at least $n$ times with $< W, n - 1 >$.
- Forcing words to be of a particular block size is space inefficient, therefore support for variable length words is possible if each word is tagged with its length. This, however, may make the plaintext more vulnerable to statistical attacks.
- In large files, $O(n)$ search time may be too expensive. This scheme can be extended to indexes, but a naïve implementation would leak a large amount of information to an attacker. Therefore, document pointers should be encrypted together with the key words, and Alice can decrypt the pointers, return the ones she wants to Bob (together with some obfuscating ones, if she wants) and he can return her the documents she wants.
- In addition, there is a trade off between time and efficiency for Alice if she wants to hide when she is updating indexes, so that an adversary cannot deduce the presence or absence of words in new documents, or edits.
- Finally, Alice can increase the security of the scheme by periodically updating the pre-encryption with a new key.