A scalable and lightweight code checker using document database

Ruo Ando[1,a)]

Abstract:
The exploitation of software vulnerability has become sophisticated. Therefore, currently safe code must obey system specific rules as well as conventional bugs such as buffer overrun. In this paper me propose a lightweight static code checker using document database. Second, a series of lexical tokens are represented as key-value form. After storing tokens, programmer write NoSQL query with domain and system specific rules for scanning vulnerability. In experiment, we apply proposed method for detecting system specific vulnerability of memory manipulation. It is shown that our approach can detect difficult-to-observe flaws with reasonable implementation cost.

1. Introduction

Software has bugs. Efficient management and prevention have been important endeavors. Unfortunately, while software has become sophisticated and diversified, system performs is ways surprising its developers. As well as safety, software vulnerabilities are an enormous cause of exploitation and Compromise in security systems. Vulnerability has become increasingly serious problem because a computer network is only as secure as its weakest link and often the software is weakest link.

From the dawn of computing era, assurance would be a One of the most critical challenge in computer security research and many research efforts have been gone into formal methods for inspecting programs. Formal approach has been proposed for discovering bugs in mission critical software and its absence of security. In many cases, push down automaton is adopted for representing security property. The model checking is applicable for identifying violation of any state desirable for security goal by checking Program reachability.

Despite these efforts, software exploitation is currently pervasive. One reason is the difficulty of coping with scalability of today's rapidly enlarging software systems. Current software is becoming large by millions of code and interacting over multiple languages and files. Secondly, we face unavoidable attributes of software development environment. Some vulnerabilities such as buffer overrun reflect poorly designed language features. Accordingly, safer languages like Java has been recommended in this aspect. However safer programming languages itself cannot prevent other vulnerabilities specifically including higher semantics. For example, system calls generated by operating system is forced to obey implicit rules on how they should be invoked. If coding errors unintentionally violate such rules in the interaction With OS, that error may introduce vulnerabilities.

Based on this fact, we propose lightweight and scalable code checker using document database. After performi.ng analysis of mature, widely deployed software, it is shown that our tool can be effective means of enhancing security of critical software at scale.

2. Background: static checking

Recently efficient software development and maintenance have become more challenging problem. Previous researches show that An experimental compile-time static inspection can discover find common programing errors with reasonable cost. Simple annotation programming language is often adopted for designing decisions which are expressed formally. [1] propose extended static checking which can be performed without running the program for finding more errors than are caught by conventional static inspection such as type checking.

Static checking takes advantages in the cost of correcting an error for improving software productivity. Particularly, if an error is reduced in the case that early detection is achieved. Figure compares three checking methods based on two aspects. Coverage and cost.] represents the degree of error coverage in running each tool and also represents its running cost. In the upper right corner is comprehensive program verification Which is designed for detecting all errors, which makes it extremely expensive. In the lower left corner are static checking method which is known as generic way. This kind of technique requires modest effort but detect an only limited range of flaw, which is the Case of conventional type checkers and type check like tool such as lint. Concerning recent computing environment, the choice of the middle of the diagram is promising. X axis also represents the decidability rate based on the fact that many factors of static checking such as array bounds errors and null dereferences is undecidableD

[1] Network Security Institute, National Institute of Information and Communications Technology 4-2-1 Nukui-Kitamachi, Koganei, Tokyo 184-8795, Japan
[a)] ruo@nict.go.jp

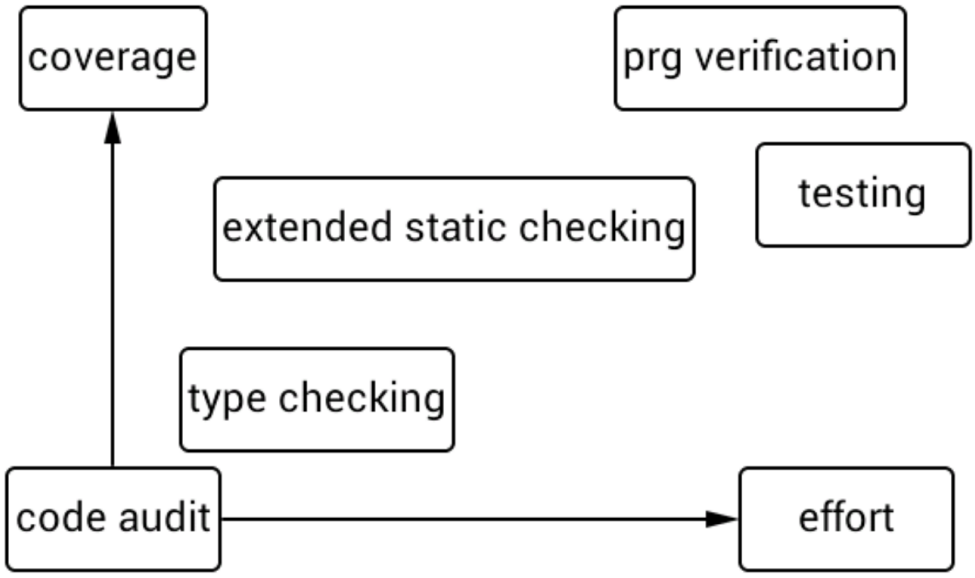

Fig. 1 Decidability for choosing checking methods.

3. Design Requirement

3.1 scalability and false negatives

In designing vulnerability checker, we face the difficult choice between precision and scalability. Particularly, security system design is forced to emphasize either false negatives or false positives. In todayfs large scale computing era, we conclude that a false negative rate should be as close to 0 as possible.

Proposed system divides a file which is not processed beforehand into u series of lexical tokens. Instead of using real parser, we define a stream of token and match it to the series. Matching code which is domain-specific representation of vulnerability is add to document database by hand. By doing this, so non-regular patterns can be identified. This method is effective for constructing more sophisticated analyzer. Furthermore, this method is easier to adapt a context-free parser with generated parse tree for navigating representation of a program. Besides, traditional Parser based analysis tool usually is not able to cope with a single build of a program at one process. part1, because a technique for parsing C programs with preprocessor directives into a comprehensive abstract syntax tree. Unfortunately, there have been no techniques proposed for parsing C and C++ with complicated directives into a single abstract syntax tree. As design requirement, the programmers would want to check every possible build of our program. Meanwhile program auditor would want to inspect the entire program without executing multiple build configurations with sufficient coverage.

In this aspect we face the trade-off between Precision and scalability. Today's open source software has become larger which requires code analyzer for scalability. Scalability should be taken into account at the stare because program analysis will have trouble handling large application eventually. Particularly, in the case of analyzing large programs such as Sendmail, apache and Linux kernel, we should aim for scalability explicitly regardless that we will some Cost in precision. Several heuristics have been proposed for trade off precision for scalability. As we give up precision, our code checker miss some implementation flaw as false negatives and could generate many false alarms as false positives. In spite of this, we believe this our tool is effective for coping with large scale software. From our experience we could conclude that relatively imprecise analysis outputs lots of false alarms, instead it can detect the number of unsafe resource operations to be inspected by hand by an order of magnitude or more.

3.2 meta rules

As systems and networks have become sophisticated, rules which system must obey are becoming complicated. While the number of conventional exploitation caused by single buffer overflow has been reduced, complicated flaw using exploitation is increasing. As a result, for securing code, programmers should obey complicated rules such as gsanitize unvalidated input before reference itg and greference permission settings before doing operation]". Every programs must obey this kind of meta rules. A single flaw allows attackers to compromise the integrity of the entire system. To make matters worse, almost all rules are difficult to understand and erastically obeyed.

It is important to point out that system specific static analysis makes it possible to discover security flaw violating meta rules 1ike "do not user assigned pointers before validating it". In previous research efforts, programmer work out system specific rules which is embedded into compiler and inspect their code. Because many security constraints are domain or even specific, hard coding a fixed set into compiler is expensive. it is preferable to cope with meta rules in the form of high-1evel, system specific checkers.

Handling hign-1evel system specific checkers takes advantages in some points. First, static inspection is able to find difficult-to-observe errors. Usually lots of security flaw is silent which is not often activated. Actually, they compromise system security with a limited range of input value while not crashing machine.

Contrary, static analysis can specify what line and what file has an error. without running code, static analysis can catch these errorsDparticularly, this feature is helpful for coping with errors in operating systems which has too many execution paths through testing. Finally, implementation cost of static analysis is reasonable. Usually, the extension is lightweight. Once the implementation fixed cost is paid, we have only pay little incremental manual cost as program size grows. Other methods like formal verification require substantial proportional to the size of code according to the size of code. Proposed system enables programmer to Implement and extend lightweight static analysis tool. we believe that many system restrictions can be represented and exploited using meta-level static analysis.

Thrust of proposed system are divided into two parts: Our checking tool can discover flaws in complicated surroundings from source code deployed in real world systems. As we discuss in experiment, our tool can handle complex situation like “memory operation under high pressures”. Besides, our tool is able to find system level flaw which are hard to detect with manual inspection. Our tool is fully corresponding t he format of document database which results in that extensions are simple.

4. proposed method

4.1 scanning and storing

In this section we describe initial scanning and storing tokens. Proposed system divided a non-preprocessed file into a series of tokens. Each token is recognized as ”key”. Then, some token are matched with ”value” assigned in the stream of tokens. In the case of processing C and C++ programs, proposed system takes one source file as input, dividing each into a stream of symbols. After scanning is completed, proposed system ’inspects the resultant token stream with traversing symbols with focusing keyword such as for, if and malloc. Selecting tokens to focus on is heuristic which is not Precise. We cannot assign every kind of tokens in C Language. Without real parsing of compiler we cannot identify all tokens that are lexically represented as symbols.

```
1
2       switch (charatyp[ch]) {
3       case Letter:
4           for ( ; charatyp[ch]==Letter || charatyp[ch]==Digit;
                ch=nextCh())
5             if (p < p_16) *p++ = ch;
6           *p = '\0';
7
8       if(strcmp(tkn.text, "memcpy")==0)
```

4.2 Backend datastore

For centralized filtering rule management, proposed system adopts data store mongoDB for storing a large number of rules rapidly while achieving scalability.

MongoDB which stands for humongous a cross-platform document oriented NoSQL database.By eschewing the traditional relational DB structure in favor of JSON-like documents which is called as BSON. By introducing BSON, MongoDB makes it possible to integrate data in certain types of applications faster and easier.

According to CAP theorem, MongoDB gives up consistency for availability and partition tolerance. CAP theorem states that a distributed system cannot satisfy these three guarantees at the same time. Consistency means that all nodes see the same data at the same time. Availability means that a guarantee that every request receives a response about whether it was successful or failed. Partition tolerance means that the system continues to operate despite arbitrary message loss or failure of part of the system.

A document-oriented database is a computer program designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Document-oriented databases are one of the main categories of NoSQL databases and the popularity of the term ”document-oriented database” (or ”document store”) has grown[1] with the use of the term NoSQL itself. In contrast to relational databases and their notions of ”Relations” (or ”Tables”), these systems are designed around an abstract notion of a ”Document”.

Listing 1 key-value.

```
1  void DBop_realloc(char *line, char *functionName, char *
       functionLine, char *filename)
2  {
3      mongoc_client_t *client;
4      mongoc_collection_t *collection;
5      mongoc_cursor_t *cursor;
6      bson_error_t error;
7      const bson_t *doc;
8      const char *uristr = "mongodb://127.0.0.1/";
9      const char *collection_name = "realloc";
10     bson_t query;
11     bson_t query_double;
12     bson_t b;
13     char *str;
14
15     mongoc_init ();
16
17     client = mongoc_client_new (uristr);
18     if (!client) {
19         fprintf (stderr, "Failed to parse URI.\n");
20     }
21
22     bson_init (&query);
23
24     bson_append_utf8 (&query, "located", -1, line, -1);
25     bson_append_utf8 (&query, "functionName", -1,
           functionName, -1);
26     bson_append_utf8 (&query, "functionLine", -1,
           functionLine, -1);
27     bson_append_utf8 (&query, "filename", -1, filename,
           -1);
28
29     collection = mongoc_client_get_collection (client, "cci",
           collection_name);
30
31     mongoc_collection_insert (collection,
32                               MONGOC_QUERY_NONE,
33                               &query,
34                               NULL,
35                               NULL);
36 }
```

4.3 data representation

In detecting vulnerabilities, proposed system reads from document database for key-value representation of focusing structures and operations. Document database keeps the entire contents for the lifetime of analysis tool. Flaw can be added to the database, removed and updated.

Document database of proposed system currently contains two major open source tree: Xen and Linux. Operations to focus are mainly memory manipulation. Basic structure of loop and branch

account for many items. For each all, proposed system the following information.

- token name.
- location. line number and function name
- range. function involved.

Listing 2
Memory

```
1
2 { "_id" : ObjectId("53d291fe40c2acf65bbbf9f7"), "
      located" : "145", "functionName" : "
      xc_vcpu_setaffinity", "functionLine" : "116", "
      filename" : "xc_domain.c" }
```

Listing 3
Loop

```
1
2 { "_id" : ObjectId("53d28a5a40c2acf65bbbf9a9"), "
      start_line" : "209", "end_line" : "250", "
      functionName" : "xc_domain_getinfo", "
      functionLine" : "197", "filename" : "xc_domain.
      c" }
3 { "_id" : ObjectId("53d28ab940c2acf65bbbf9b1"), "
      start_line" : "212", "end_line" : "253", "
      functionName" : "xc_domain_getinfo", "
      functionLine" : "200", "filename" : "xc_domain.
      c" }
```

Unfortunately, the document database has several limitations. For example modified code about how to mitigate the problem is desirable.

- detailed decriptions of guessed vuluerability
- the recommendations should be added.
- several additional fields would be preferable such as callchain between variables.

Currently interface of our database has command line interface of python. The programmer is able to write routines for which proposed system should check specifically.

5. experiment

5.1 realloc vulnerability

Use-after-free vulnerability in the libxl_list_cpupool function in the libxl toolstack library in Xen 4.2.x and 4.3.x, when running "under memory pressure," returns the original pointer when the realloc function fails, which allows local users to cause a denial of service (heap corruption and crash) and possibly execute arbitrary code via unspecified vectors. http://www.cvedetails.com/cve/CVE-2013-4371/

The function is located at:

```
# global -t  libxl_list_cpupool
libxl_list_cpupool       tools/libxl/libxl.c      388
```

at line 402, Xen uses realloc for reallocating the memory. Note that the address of libxl_cpupoolinfo is already assigned outside of this routine. Under high pressure, realloc can not extend the memory from the original pointer which is already obtained. in this case, realloc newly yielding the address which remaining the data to be written.

Listing 4
Realloc

```
1
2  402 tmp = realloc(ptr, (i + 1) * sizeof(libxl_cpupoolinfo));
3
4  388libxl_cpupoolinfo * libxl_list_cpupool(libxl_ctx *ctx, int *
       nb_pool)
5  389{
6  397 poolid = 0;
7  398 for (i = 0;; i++) {
8  399 info = xc_cpupool_getinfo(ctx->xch, poolid);
9  400 if (info == NULL)
10 401 break;
11 402 tmp = realloc(ptr, (i + 1) * sizeof(libxl_cpupoolinfo));
12 403 if (!tmp) {
13 404 LIBXL__LOG_ERRNO(ctx, LIBXL__LOG_ERROR, "
       allocating cpupool info");
14 405 free(ptr);
15 406 xc_cpupool_infofree(ctx->xch, info);
16 407 return NULL;
17 408 }
18 409 ptr = tmp;
19 410 ptr[i].poolid = info->cpupool_id;
20 411 ptr[i].sched_id = info->sched_id;
21 412 ptr[i].n_dom = info->n_dom;
22 413 if (libxl_cpumap_alloc(ctx, &ptr[i].cpumap)) {
23 414 xc_cpupool_infofree(ctx->xch, info);
24 415 break;
25 416 }
26 417 memcpy(ptr[i].cpumap.map, info->cpumap, ptr[i].
       cpumap.size);
27 418 poolid = info->cpupool_id + 1;
28 419 xc_cpupool_infofree(ctx->xch, info);
29 420 }
```

The realloc() function shall change the size of the memory object pointed to by ptr to the size specified by size. The contents of the object shall remain unchanged up to the lesser of the new and old sizes.

after failing realloc, pointer is freed and xc_cpupool_infofree(ctx-¿xch, info) is invoked. it seem that the code is already pathed, though.

The realloc funcition copes with heap and is usually used for increasing the size of a block of allocated memory. Realloc often needs copying contents of the old memory block into a new and larger block. Realloc leaves the data of the original block input but inaccessible to the process, preventing the program from being able to scrub sensitive data from memory. If an attacker is able to later examine the contents of a memory dump, the sensitive data could be exposed.

5.2 results

Listing6 shows brief output of detecting heavy memory manipulation. in this result, realloc is located at line 402. The realloc invocation is located at function which starts at line 388. Second item represents information of forloop written in libxl.c. Iteration starts at 388 and ends at 420.

Listing 5
Main loop

```
1
2 { "_id" : ObjectId("53f9ec4764e21cef244d69fb"), "
      located" : "402", "functionName" : "
      libxl_list_cpupool", "functionLine" : "388", "
      filename" : "libxl.c" }
3
4 { "_id" : ObjectId("53f9ec9464e21cef244d6a0e"), "
      start_line" : "398", "end_line" : "420", "
      functionName" : "libxl_list_cpupool", "
      functionLine" : "388", "filename" : "libxl.c" }
```

6. Related Work

A number of research efforts have been proposed on detecting

Software vulnerabilities and configuration flaws. Integer range analysis has been proposed for identifying buffer overruns [1]D They have developed a tool for static detection of buffer overflow in C programs. It is shown that their tool is effective for discover both known and unknown buffer overflow vulnerabilities in sendmail. Integer range analysis problem is a formulation of the problem for testing C strings as abstract type accessed through library functions and modeling pointers. Their system can handle constraints which is similar to Lint[2] for operations involving strings.

Three key analysis techniques of context sensitive, flow sensitive and interprocedual data flow are adopted. Kored et al for computing access rights requirement in Java. This method allows optimizations to keep the analysis tractable[3]. For inspecting properties of C programs, CQAUL[4] is proposed with type based analysis. authorization of hook placement of Linux security Model Framework is proposed in[5].

It has been used to detect format string vulnerabilities [6] and to verify authorization hook placement in the Linux security Model Framework. which are examples of the development of sound analysis for verification of particular security properties.

Unix grep is simple but frequently used for finding potentially unsafe library function calls. For finding some Program flaws by lexical analysis, ITS4[7] points out some limitations of using grep.ITS4 is proposed for challenging this problem with lexical analysis tool which finds a security problems using a potentially risky conflicts. Adopted lexical analysis are fast and simple. However, its effectiveness is limited int he point that ITS4 does not take into account the syntax or semantics of the program. For a deeper inspection, more precise checking is necessary.

Metal[8] is early research effort for checking rule violation in order to write system-specific compiler extensions. Metal is basically designed for programmer/security analyst to cope With specify annotations on the target program.

SLAM [9] has been milestone project which leverage software model checking for verification of temporal safe properties, SLAM show the direction of formal verification. However. SLAM is not supporting analysis of large programs,

MOPS[10] is a model checking based tool or inspecting security critical applications. If a security property is defined, Mops detects the violation of the property. For example, Mops can check the violations of folk rules for writing secure programs especially for setuid programs. MOPS provides the ability to output at one error trace for each error. By doing this, the usability is improved by reducing the number of error trace to be reviewed

Ann Despite important technical differences, they believe these diverse approaches to software model checking share significant common ground, and we expect that much of our implementation experience would transfer over other tools annotations and libraries. Usually, constraints are not generated from annotations. Constraints are generated for standard library functions which is embedded. Flow incentive analysis is adapted to cope with the constraints. Flow sensitive analysis which can be scaled for handling real programs requires the localization provided by annotations. In general, flow incentive analysis is not accurate enough to enable special handling of loop or if statements.

Another few tools have been introduced to array bound vulnerability in C language. [11] propose an extended static checking adopts an automatic theorem prover to find array index bounds error. Another extended static checking leverages information provided by annotations to assist checking.

These diverse methods for vulnerability checking provides significant common ground. We expect that our implementation experience could transfer over other methods.

7. Conclusions

The exploitation of software vulnerability has become sophisticated. Therefore, safe code must obey system specific rules as well as conventional bugs such as buffer overrun. In this paper me propose a lightweight static code checker using NoSQL. First, proposed system divides program code into tokens. Second, a series of lexical tokens are represented as key-value form. After storing tokens, programmer write NoSQL query with domain and system specific rules for scanning vulnerability. in experiment, we apply proposed method for detecting system specific vulnerability of memory manipulation. It is shown that our approach can detect difficult-to-observe flaws with reasonable implementation cost.

References

[1] David Wagner, Jeffrey S. Foster, Eric A. Brewer, Alexander Aiken: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. NDSS 2000

[2] Stephen Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977.

[3] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2002.

[4] J. Foster, M. FNahndrich, and A. Aiken. A theory of type quali?ers. In ACM SIGPLAN Conference on Program- ming Language Design and Implementation (PLDIf99), May 1999.

[5] Antony Edwards, Trent R. Jaeger, Xiaolan Zhang, Verifying Authorization Hook Placement for the Linux Security Modules Framework, Proceedings of the 9th ACM Conference on Computer and Communications Security. , ACM. , 225-34 in 2002

[6] Detecting Format String Vulnerabilities with Type Qualifiers. U Shankar, K Talwar, JS Foster, D Wagner USENIX Security Symposium, 201-220, 423, 2001

[7] John Viega Bloch, ITS4: a static vulnerability scanner for C and C++ code, Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference

[8] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In Proceed- ings of IEEE Security and Privacy 2002, 2002

[9] T. Ball and S. K. Rajamani. The SLAM project: Debug- ging system software via static analysis. In POPL f02: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 2002.

[10] H. Chen and D. Wagner. MOPS: an infrastructure for ex- amining security properties of software. In Proceedings of the 9th ACM Conference on Computer and Communica- tions Security (CCS), Washington, DC, 2002

[11] Flanagan, Leino, Lillibridge, Nelson, Saxe, State, Extended Static Checking for Java, PLDI 02

[12] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In POPL f03: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 2003.

[13] H. Chen, D. Wagner, and D. Dean. Setuid demysti?ed. In Proceedings of the Eleventh Usenix Security Symposium, San Francisco, CA, 2002.

[14] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive pro- gram veri?cation in polynomial time. In PLDI f02: Pro- ceedings of the ACM SIGPLAN 2002 Conference on Pro- gramming Language Design and Implementation, Berlin, Germany, June 2002.

[15] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules

using system-speci?c, programmer-written compiler extensions. In OSDI, 2000.

[16] D. Engler and M. Musuvathi. Static analysis versus soft- ware model checking for bug ?nding. In 5th Intl. Confer- ence Veri?cation, Model Checking and Abstract Interpre- tation (VMCAI f04), 2004.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software veri?cation with BLAST. In Proceedings of the 10th SPINWorkshop on Model Checking Software, 2003.