International Journal on New Computer Architectures and Their Applications (IJNCAA) 1(3): 652-664
The Society of Digital Information and Wireless Communications, 2011 (ISSN: 2220-9085)

# Classification and Measurement on C Overflow Vulnerabilities Attack

Nurul Haszeli Ahmad[1,2], Syed Ahmad Aljunid[2], Jamalul-lail Ab Manan[1]
[1] MIMOS Berhad, TPM Bukit Jalil, 57000 Kuala Lumpur, Malaysia
[2] Faculty of Computer Sciences and Mathematics, UiTM, Shah Alam 40000, Selangor, Malaysia
{haszeli.ahmad, jamalul.lail}@mimos.my, aljunid@tmsk.uitm.edu.my

## ABSTRACT

Since early 70s, softwarrrre vulnerabilities have been classified and measured for various purposes including software assurance. Out of many software vulnerabilities, C vulnerabilities are the most common subject discussed, classified and measured. However, there are still gaps in those early works as C vulnerabilities still exist and reported by various security advisors. The most common and highly ranked is C overflow vulnerabilities. Therefore, we propose this taxonomy, which classified all existing overflow vulnerabilities including four vulnerabilities that have never been classified before. We also provide a guideline to identified and avoid these vulnerabilities from source code perspective. We ensure our taxonomy is constructed to meet the characteristics of well-defined taxonomy. We also evaluate our taxonomy by classifying various software security advisories and reports using our taxonomy. As a result, our taxonomy is complete and comprehensive, and hence, is a valuable reference to be used as part of software assurance processes.

## KEYWORDS

Taxonomy, Classification, Buffer Overflow, Source Code Vulnerabilities, Software Security, Exploitable Vulnerability.

## 1 INTRODUCTION

Since the first recorded vulnerabilities exploitation [1], with various protection and preventive mechanism developed and enhanced, C vulnerabilities exploitation is still a huge issues in software security community [6], [7], and [8]. From numerous C vulnerabilities, overflow vulnerabilities was identified as the most crucial as it is still the most dominant and ranked with high severity [9], [10], [11], [15], [16], [17] and [19]. No doubt that previous work has significant impact in reducing C vulnerabilities. However, there are still improvements needed to eliminate the issue or at least minimize the possibility of C vulnerabilities from occurring.

Through our analysis on works by [2], [3], [4], [5], [12], [13], and [14], we conclude that there are three major categories of improvement; vulnerability understanding, analysis tool, and security implementation. For this purpose, we limit our discussion to vulnerability understanding since accurate comprehension on the matter is crucial to improve analysis tool and security implementation, as shared by [18]. In this paper, we synthesize and construct a taxonomy focusing on C overflow vulnerabilities since there is no taxonomies addressing C overflow vulnerabilities from source code perspective. We also describe each behavior, structure, and rules to find and avoid these vulnerabilities. In addition, we also construct experiments to verify the possibility of these vulnerabilities to

occur in current operating system i.e. Windows XP and Windows Seven.

## 2 PREVIOUS WORKS

Various taxonomies have been constructed and presented ranging from numerous perspectives, scopes and purposes. Despite differences, those taxonomies share the same objective; to minimize exploitable software vulnerabilities. [18], [19], [20], [21], [22], [23], [24], [25], [26], [28], and [29] presented general vulnerability taxonomies whereas [2], [3], [4], [32], and [33] focused on C overflow vulnerabilities.

While those past works on taxonomies have significant impact in reducing vulnerabilities and exploitation, renown security institutes and corporations [8], [10], and [11] continue issuing reports and advisories on C overflow vulnerabilities, signifying breaches for exploratory discovery to aim for superior community comprehension of C overflows vulnerabilities. Most of these taxonomies were subsequently reviewed and analyzed, as done by [30]. Our work however focuses on C overflows vulnerabilities. As such, taxonomies that do not enclose or discuss C overflow vulnerabilities are ignored. Table 1 summarized our study on previous taxonomies focusing on types of C overflow vulnerabilities.

**Table 1.** Summary of Previous Vulnerabilities Taxonomies

| Author | Type of C Overflows Vulnerabilities | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | UF | AO | IO | RIL | MF | FP | VTC | PS | UV | NT |
| [31] | **Ø** | √ | × | × | × | × | × | × | × | × |
| [21] | × | √ | × | × | × | × | × | × | × | × |
| [26] | **Ø** | √ | √ | × | **Ø** | √ | × | × | √ | √ |
| [27] | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ | ≈ |
| [33] | **Ø** | √ | √ | × | × | × | × | × | × | × |
| [32] | **Ø** | √ | √ | × | × | × | × | × | × | × |
| [4] | **Ø** | √ | × | × | × | × | × | × | × | × |
| [3] | **Ø** | √ | × | × | × | × | × | × | × | × |
| [2] | **Ø** | √ | × | × | × | × | × | × | × | × |

* Notation

UF – Unsafe Function
AO – Array Out-of-bound
IO – Integer Overflow
RIL – Return-Into-LibC
MF – Memory Function
FP – Function Pointer / Pointer Aliasing
VTC – Variable Type Conversion
PS – Pointer Scaling / Mixing
UV – Uninitialized Variable
NT – Null Termination

**Ø** – Partially Classified
√ - Classified
× - Not Classified
≈ - Generally mention

As shown in Table 1, we discover four new types of overflow vulnerabilities necessitate classification i.e. Unsafe Function, Return-Into-LibC, Memory Function and Variable Type Conversion. We do not consider Pointer Scaling/Mixing as new overflow vulnerabilities type as it was classified in Function Pointer/Pointer Aliasing group by [26]. However, in our taxonomy, we separate those two due to different behavior and method used to identify the types and it is further explained in section 3.8.

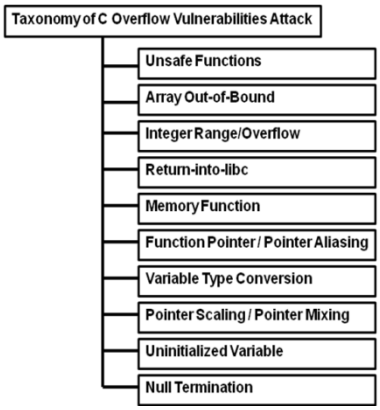
## 3 TAXONOMY OF C OVERFLOW VULNERABILITIES ATTACK

We evaluate sets of vulnerabilities advisories and exploitations reports since 1988 until 2011. There are more than 50000 reported cases of C overflow vulnerabilities originating from five

vulnerabilities databases and malware collection sites [9], [34], [6], [7], and [35].

From these reports, we classify types of C overflow vulnerabilities into ten categories. Four of them are new and still unclassified. These are unsafe functions, return-into-libc, memory functions and variable type conversion. They have at least a medium level of severity, possibility to appear and exploited [6], [7], and [9]. The impact of exploitation with unsafe function is recognized as the most dangerous and outstanding [9], [34], [6], [7], and [35].

Figure 1 visualizes the new taxonomy of overflow vulnerability attack, organized




in accordance to its severity, dominance, potential occurrence and impact. This taxonomy simplifies the understanding on implications of each types, their behavior and preventive mechanisms.

**Figure 1.** Proposed Taxonomy for Overflow Vulnerabilities Attack in C

### 3.1 Unsafe Functions

Although unsafe functions have been exploited since 1988 [1], [15], 17], it is still relevant. More importantly, this well-known and well-documented inherent C security vulnerability is categorized as the most critical software vulnerabilities to continue to dominate C vulnerabilities report [6], [7], [35] and [39]. This implies that there are software developers who are either ignorant, unaware, or simply bypass software security policies for prompt development [15], [16]. Below is a sample of unsafe functions vulnerability.

Part of a program showing *scanf()* vulnerability.

```
 …
char str[20];
char str2[10];

scanf("%s",&str);
scanf("%s",&str2);
…
```

By supplying an input greater than the allocated size at the first *scanf()*, it automatically overflows the seconds variable and force the program to skip the second *scanf()*. This is one of many unsafe functions in C [12], [15], [16], [17], [36], [37] and [38]. Previous taxonomies classified few unsafe functions as Format String Attack, Read-Write, or Buffer-Overrun [2], [3], [15]. This is arguable since there are unsafe functions that do not implement formatting or require specifying index for reading or writing.

To prevent overflows via unsafe functions, one needs to check input variable before passing into any unsafe functions. Alternatively, there is C library safe functions that developers can use to avoid this type of overflow [17], [37].

### 3.2 Array Out-of-bound

Array Out-of-bound overflow can be triggered by misuse or improper handling of an array in a read or write

operation, irrespective of it being above upper or below lower bound. A true sample is shown below.

A section from linux driver code in i810_dma.c contains the vulnerability [40], [41].

```
if(copy_from_user(&d,        arg,
sizeof(arg)))
      return -EFAULT;
if(d.idx > dma->buf_count)
      return -EINVAL;
buf     =     dma->buflist[d.idx];
//overflow if d.idx == -1
copy_from_user(buf_priv-
>virtual, d.address, d.used);
```

As shown in the above sample, when *d.idx* contains the value of -1, it will bypass the conditional statement which triggers overflow on the following statement. Array Out-of-bound overflows is easy to detect and prevent by monitoring all array processes and verifying whether the index is within the range specified; between zeros to less than one from total array size.

### 3.3 Integer Range / Overflow

This type of overflow may occur due to miscalculation or wrong assumption in an arithmetic operation and is gaining its popularity in vulnerabilities databases [42], [43], [44]. The possibility of exploit is small, but the result of exploiting it is significantly dangerous [45].
This classification is attributed from [26], [32], and [33]. The key difference is the removal of numerical conversion as one of the integer overflow type, and classifies it in a different category. This is due to its differences in behavior and code structure. Furthermore, the conversion errors are dependent on platform used to execute it. A true sample from [45] is shown below.

A fraction of C code contains Integer Range/Overflow vulnerability [45].

```
nresp = packet_get_int();
if (nresp > 0) {
response =
xmalloc(nresp*sizeof(char*));
  for (i = 0; i > nresp; i++)
   response[i] =
   packet_get_string(NULL);
}
```

As shown in the above code, if one able to inject input causing variable *nresp* to contain large integer, the operation *xmalloc(nresp*sizeof(char*))* will possibly trigger overflow, and later can be exploited [45]. It is difficult to detect as one needs to understand the logics and predict possible outcome from the arithmetic operation. As a result, this vulnerability tends to be left out undetected either by analysis tool or manual code review. This vulnerability can be avoided by simply restricting the possible input value before arithmetic operation took place.

### 3.4 Return-into-libC

Although it has been recognized as earlier as unsafe functions [84], it is yet to be appropriately classified. Many vulnerabilities databases rank its severity as high although the number of occurrence is low. It is difficult to detect since it can only appear during runtime and the code itself does not have specific characteristic to indicate it as vulnerable. Earlier protection tools such as ProPolice and StackShield have failed to detect [46]. It is also difficult to exploit since ones need to know the exact length

of character, address of function call, and address of environment variable.

A sample vulnerable code contains return-into-libc vulnerability.

```
int main(char **av)
{
    char a[20];
    if ( strlen(av[1]) < 20 )
//verified the length
       strcpy(a, av[1]);
//nothing happen
    printf ("%s", a);
//possible have vulnerability
    return 0;
}
```

Based on the code above, it is possible to supply a string long enough to fill up the allocated memory space together with function call to replace the defined return address. The result of exploiting it is extremely dangerous [47]. To recognize the vulnerability, security analysts need to understand possible input values and estimate memory location. It is similar to Unsafe Function and Array Out-of-bound class but differ in terms of behavior and memory process. Memory in the former two classes will overflow and overwrite the return address, resulting in unintended behavior. In contrast, Return-into-lib will replace return address with a function call to another program e.g. *system()* and *WinExec()* [48], [49], [50]. To prevent it from appearing or being exploited, the contents of the input must be validated apart from the length.

### 3.5 Memory Function

Even though it has been in the security radar as early as 2002 [52], [53], [54], [55], [56], [57], [58], [59], it is not been properly classified. This type of vulnerability has gain notoriety as one of the preferred vulnerability for exploitation due to current programming trend which utilizes dynamic memory for better performance and scalability.
Double call on *free()* function, improper use of *malloc(), calloc(),* and *realloc()* functions, uninitialized memory, and unused allocated memory are few examples of memory functions vulnerabilities. Simple memory function vulnerability is shown below.

A fragment of C code with *free()* function vulnerability.

```
char* ptr = (char*) malloc
(DEFINED_SIZE);
...
free(ptr);    //first call to
free ptr
free(ptr);    //vulnerable due
to free the freed ptr
```

As shown above, the second call to free the same variable will cause unknown behavior. This can be used for exploitation and its severity is equivalent to the first three types [61], [60].
Due to its safe nature and programming complexity, it is difficult to assess its vulnerability potential unless an in-depth semantics view of program is used. From coding perspective, this type of vulnerability can be prevented by validating the memory before usage, initializing memory with default value depending on variable type, and removing unused memory.

### 3.6 Function Pointer / Pointer Aliasing

Function pointer or pointer aliasing is a variable storing address of a function or as reference to another variable. It can

later be called by the given pointer name which assists developer's flexibility and ease of programming. It becomes vulnerable when the reference has been nullified or overwritten to point to a different location or function [52], [62], [63], [64].

It is difficult to detect by manual code review unless it is done by highly experience security analysts. However, using automatic tool requires the tool to comprehend semantically the code [65]. Below is an example.

An example of pointer aliasing vulnerability.

```
char s[20], *ptr, s2[20];
ptr = s;
//vulnerable line of code
strncpy(s2, ptr, 20); //
vulnerability realized
```

As shown above, the pointer *ptr* is referring to a null value since the variable *s* is yet to be initialized. The subsequent line of code realizes the vulnerability, although the function used is a safe function. The only way to stop this type of vulnerability from continuing to occur is by enforcing validation on pointer variable before being used throughout the program.

### 3.7 Variable Type Conversion

Improper conversion of a variable can create vulnerability and exploitation [67], [68], [69]. Although there are considerable numbers of advisories reporting this vulnerability [67], [70], it was never mentioned in any earlier taxonomy. It may be due to infrequent occurrence and minimal severity. It was also considered as a member of integer overflow vulnerability which is arguable since conversion errors do happen on other data format. A true example of this vulnerability is shown below.

Fraction of Bash version 1.14.6 contains Variable Type Conversion vulnerability [73].

```
static int yy_string_get() {
  register char *string;
  register int c;

  string                        =
bash_input.location.string;
  c = EOF;

  ......
  if (string && *string) {
    c       =       *string++;
//vulnerable line
    bash_input.location.string
= string;
  }
  return (c);
}
```

This vulnerability is proven to be exploitable [67], [68], [69], [71], and [72]. Ignoring it is reasonably risky. To avoid conversion error vulnerability, it is strongly suggested to validate all variable involves in conversion, as well as avoid unnecessary conversion, or use the same data type.

### 3.8 Pointer Scaling / Mixing

This vulnerability may arise during an arithmetic operation of a pointer [74], [75]. Semantically, it is different to pointer aliasing in terms of coding. It seldom happens but the impact of exploiting it is comparable to other type of overflow vulnerability.

In a pointer scaling or mixing process, the size of object pointed will determine the size of the value to be added [75]. If one failed to understand this, he or she may wrongly calculate and assign the

wrong size of object to accept the result, and therefore runs the risk of having overflow vulnerability.

Sample of code contains Pointer Scaling / Mixing vulnerability [76].

```
int *p = x;
char * second_char = (char
*)(p + 1);
```

As shown in the above code, subsequent read or write to pointer *second_char* will cause overflow or unknown behavior due to addition of value 1 to current address location of variable *x*.
To avoid this vulnerability from occurring, ones must recognize and be able to correctly determine the accurate size of recipient variable and actual location in memory.

### 3.9 Uninitialized Variable

Uninitialized variable is a variable declared without value assigned to it. Nonetheless, computer will allocate memory and assign unknown values, which later if being used will cause computer system to perform undesired behavior [77], [78], [79]. It can also be exploited by attackers thus allowing the system to be compromised [80].

A fraction of C code contains Uninitialized Variable vulnerability [80].

```
....
void take_ptr(int * bptr){
      print ("%lx", *bptr);
}

int main(int argc, char **argv){
      int b;
      take_ptr(&b);
      print ("%lx", b);
}
```

Variable *b* in the sample above was not initialized and then being used twice without value assigned to it. By default, a location has been set in memory and the next line after declaration will force either computer system to behave abnormal or be vulnerable for exploitation. Normal compiler, code review, or even most static analysis will not mark this as vulnerable. There is also possibility of triggering false alarm if there is value assign to it before use.
The easiest method to overcome this vulnerability is to initialize all variable with acceptable default value such as zero for numerical type of variable and empty string or blank space for character or string. It must not be left uninitialized or contain null value before used. Another approach to avoid this vulnerability which might impact performance is to validate variable before usage.

### 3.10 Null Termination

Although it seem likely to ensue and can easily be avoided, it still appear in few vulnerability databases [82] and [83]. The consequence of having this vulnerability is equally hazardous as other type of vulnerabilities [81].
Null termination vulnerability is defined as improper string termination, array that does not contain null character or equivalent terminator, or no null byte termination with possible impact of causing overflows [52], [54], [81]. A sample of this vulnerability is shown in code below.

Fraction of C code contains Null Termination vulnerability [81].

```
#define MAXLEN 1024
...
```

```
char *pathbuf[MAXLEN];
...
read(cfgfile,inputbuf,MAXLEN);
strcpy(pathbuf, inputbuf);
```

The above code which seems safe as the *read()* function has limited the size of input to the same size of destination buffer on the last line. However, if the input did not have null termination, due to behavior of *strcpy()*, it will continue to read it until it find a null character. This makes it possible to trigger an overflow on the next reading of memory. Even if it was replaced with *strncpy()*, which is considered as safe, the behavior is still unpredictable, thus making it a unique vulnerability on its own.
To overcome the vulnerability, one should validate the input before use, or restrict the length of input to have less than one from the actual defined size.

## 4 TAXONOMY EVALUATIONS

We conduct two experiments to evaluate our taxonomy effectiveness and comprehensiveness. For the first experiment, we select five developers whom are familiar with C Language and present our taxonomy to them. Those five developers were requested to map security advisories related to C overflow vulnerabilities with the given taxonomy. We run the evaluation in three phases. In the first phase, each developer is given set of reports and taxonomy and no explanation given in using the taxonomy. On phase two, we provide explanation and guide them using the same set of reports. On phase three, they are given a new set of reports. The result shown in the table below is based on the last phases of the evaluation.

**Table 2.** Evaluating Taxonomy for Effectiveness and Comprehensiveness

| Tester | Result | | | | |
|---|---|---|---|---|---|
| | No. Report | S | M | F | SR |
| 1 | 100 | 90 | 4 | 6 | 0.9 |
| 2 | 100 | 78 | 7 | 15 | 0.78 |
| 3 | 100 | 96 | 0 | 4 | 0.96 |
| 4 | 100 | 85 | 8 | 7 | 0.85 |
| 5 | 100 | 92 | 1 | 7 | 0.92 |

* Notation
S – Successful Mapping
M – Mismatch
F – Failed to match
SR – Successful rate (%)

Based on the result on table 4, it is concluded that our taxonomy has an average of 0.882 successful rates in identifying C overflow vulnerabilities. The rates can be improved by providing few samples and detail explanation of each type of overflow vulnerabilities.
The second experiment was to evaluate comprehensiveness of our taxonomy and relevancies as of today environment. Our scope is limited to 32-bit operating systems since it is widely used by computer user compare to 64-Bit which is mostly used in server environment. We created few programs for each type of overflow vulnerabilities and executed those programs in three types of operating system. The experiment was conducted for three times daily for one week. For this purpose, we used a personal computer (PC) with 4GB RAM and processor Intel Pentium Core 2 Duo. The PC was installed with three operating system; Windows XP Professional (Service Pack 3) 32Bit, Windows 7 Professional 32Bit, and Linux Centos 5.5 32Bit. Each program was compiled using compiler MinGW GCC 32bit. Result of our experiment is tabularized below.

**Table 3.** Evaluating Taxonomy for Comprehensiveness and Relevancies

| Overflows Types | Result | | |
|---|---|---|---|
| | Windows XP SP3 | Windows 7 | Linux Centos 5.5 |
| Unsafe Functions | √ | √ | √ |
| Array Out-of-bound | √ | √ | √ |
| Integer Range/Overflow | √ | √ | √ |
| Return-Into-LibC | Ø | Ø | Ø |
| Memory Function | Ø | √ | Ø |
| Function Pointer / Pointer Aliasing | Ø | Ø | Ø |
| Variable Type Conversion | Ø | Ø | Ø |
| Pointer Scaling / Mixing | Ø | Ø | Ø |
| Uninitialized Variable | Ø | Ø | Ø |
| Null Termination | Ø | Ø | Ø |

* Notation
**Ø** – Partially Overflow × - No possibility of overflow
√ - Overflow

As shown in Table 5, the first three dominant and severe types of overflow vulnerabilities are still relevant and have the possibility of occurring again if there is no essential action taken to eliminate those vulnerabilities. This also implies that although Unsafe Functions, Array Out-of-bound and Integer Overflow are common, well-defined vulnerabilities do exist in various security reports [6], [7], [35] and [39]. There is still deficient of security concern among developers. Added to those, three is Uninitialized Variable which can be easily avoided if software developers regard the importance of initializing variable.

For other types of overflow vulnerabilities, we consider them as partial overflow as they are depending on few conditions such as code complexity, functions being used and type of variable used. However, we cannot ignore the possibility of those vulnerabilities to appear and thus it is still relevant until today.

## 5 SUMMARY OF TAXONOMY ON C CODE OVERFLOW VULNERABILITIES ATTACK

We have presented our taxonomy and briefly explain on each of the categories in our taxonomy, method to identify and avoid those vulnerabilities and evaluate our taxonomy via two experiments. As summary of our taxonomy, we tabulate below the types or categories or overflow vulnerability attacks, technique it being manipulated or exploited, method to identify, the severity, occurrences, and probability of reappearing. The occurrence and severity of listed vulnerability type is based on our thorough evaluation on various advisories and reports by [7], [8], [10], and [11] whereas the probability is based on our experiment.

**Table 4.** Summary of Taxonomy on C Code Overflow Vulnerability Attack

| Overflow Type | MOE | CA | S | O | P |
|---|---|---|---|---|---|
| Unsafe Function | Supplying malicious input long enough to overwrite memory location | No validation on input before being used in unsafe function or restricting unsafe function | C | H | H |
| Array Out-of-Bound | Supplying input or forcing access on array beyond defined index either | No validation on index of array before being used. | C | H | H |

| Overflow Type | MOE | CA | S | O | P |
|---|---|---|---|---|---|
| | below minimum or above minimum index. | | | | |
| Integer Range/Overflow | Supplying input used in arithmetic operation forcing the result to overwrite memory defined or exploiting miscalculation of arithmetic operation | Improper estimation on result of arithmetic calculation | C | H | H |
| Return-into-libc | Overwriting return address with address of library function | Uncheck argument passing in a function call | C | L | L |
| Memory Function | Exploiting misuse of memory function (i.e. double call to *free()*) | Never use allocated memory, double free of same memory or calling freed memory. | C | M | M |
| Function Pointer / Pointer Aliasing | Overwriting the function pointer to point address that contains malicious code or function | Use of pointer without validating the pointer first | M | M | M |
| Variable Type Conversion | Exploiting vulnerabilities exist during conversion of different variable type | Miscalculation of variable size involves in conversion | M | L | M |
| Pointer Scaling / Pointer Mixing | Exploiting vulnerabilities trigger during arithmetic operation of a pointer | Miscalculation of pointer size in scaling or mixing process | M | L | M |
| Uninitialized Variable | Exploiting vulnerabilities when uninitialized variable being used in the program | A variable being used before initialization | M | L | L |
| Null Termination | Supplying non-terminated input | No null termination validation on input | M | L | M |

* Notation
MOE – Mode of Exploit
CA – Code Appearance
S – Severity
O – Occurences
P – Probability of occurring
C – Critical
M – Medium
H – High
L - Low

## 6 CONCLUSIONS

We have discussed various classifications of software overflow vulnerabilities, and presented the strengths and weaknesses of previous taxonomies in general, and overflow and C vulnerabilities in particular. We noted at present there is no taxonomy specifically addressing overflow vulnerabilities from C source code perspective. Therefore, we construct taxonomy for C overflow vulnerabilities attack. In producing this taxonomy, we focus on how the overflow vulnerability appears in C code and the criteria used for a code to be considered as vulnerable. We demonstrated the application of our taxonomy in identifying types of C overflow vulnerabilities by providing a few sample vulnerable code segments. The taxonomy can be a valuable reference for developers and security analysts to

identify potential security C loopholes so as to reduce or prevent exploitations altogether. We also evaluate our taxonomy on its effectiveness, comprehensiveness, and relevancies to prove the important of having our taxonomy as part of understanding and eliminating C overflows vulnerabilities.

## 7 FUTURE WORKS

We look forward to extend our validation and verification of our taxonomy with standard vulnerability databases to large set of developers and implement it to evaluate the effectiveness of the security vulnerability program analysis tools.

## 8 REFERENCES

1. Aleph One: Smashing the Stack for Fun and Profit. Phrack Magazine. Volume 7, Issue 49, (1996)
2. Zitser, M.: Securing Software: An Evaluation of Static Source Code Analyzers. M. Sc. Thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2003)
3. Kratkiewicz, K.: Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. M. Sc. Thesis. Harvard University (2005)
4. Zhivich, M. A.: Detecting Buffer Overflows Using Testcase Synthesis and Code Instrumentation. M. Sc. Thesis. Massachusetts Institute of Technology (2005)
5. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing Memory Error Exploits with WIT. In: IEEE Symposium on Security and Privacy, pp. 263 -- 277. IEEE Computer Society Washington, DC, USA (2008)
6. Common Vulnerability and Exposures, http://cve.mitre.org/
7. Microsoft Security Advisories, http://www.microsoft.com/technet/security/advisory
8. IBM X-Force Threat Reports, https://www-935.ibm.com/services/us/iss/xforce/trendreports/
9. 2010 CWE/SANS Top 25 Most Dangerous Software Errors, http://cwe.mitre.org/top25/
10. Buffer Overflow on Common Vulnerability and Exposures, http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Buffer+Overflow
11. Microsoft Security Advisories Archive, http://www.microsoft.com/technet/security/advisory/archive.mspx
12. Chess, B., McGraw, G.: Static Analysis for Security. J. IEEE Security and Privacy. Volume 2. Issue 6. 76 -- 79 (2004)
13. Foster, J. S., Hicks, M. W., Pugh, W.: Improving software quality with static analysis. In: 7th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for software tools and engineering, pp. 83 -- 84. ACM, New York (2007)
14. Emanuelsson, P., Nilsson, U.: A Comparative Study of Industrial Static Analysis Tools. J. Electronic Notes in Theoretical Computer Science (ENTCS). Volume 217. 5--21 (2008)
15. Howard, M., LeBlanc, D., Viega, J.: 24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. McGraw Hill, United States of America (2009)
16. Viega, J., McGraw, G.: Building Secure Software: How to Avoid Security Problems the Right Way. Addison-Wesley Professional, United States of America (2001)
17. Seacord, R. C.: Secure Coding in C and C++. Addison-Wesley Professional, United States of America (2005)
18. Krsul, I. V.: Software Vulnerability Analysis. Phd. Thesis. Purdue University (1998)
19. Lough, D. L.: A Taxonomy of Computer Attacks with Applications to Wireless Networks. Phd. Thesis. Virginia Polytechnic Institute and State University (2001)
20. Aslam, T.: A Taxonomy of Security Faults in the UNIX Operating System. M. Sc. Thesis. Department of Computer Science, Purdue University (1995)
21. Alhazmi, O. H., Woo, S. W, Malaiya, Y. K.: Security Vulnerability Categories in Major Software Systems. In: 3rd IASTED International Conference on Communication, Network, and Information Security (CNIS). ACTA Press, Cambridge, USA (2006)
22. Pothamsetty, V., Akyol, B.: A Vulnerability Taxonomy for Network Protocols: Corresponding Engineering Best Practice Countermeasures. In: IASTED International Conference on Communications, Internet, and Information Technology (CIIT). ACTA Press, US Virgin Islands (2004)
23. Bazaz, A., Arthur, J. D.: Towards A Taxonomy of Vulnerabilities. In: 40th International Conference on System Sciences. Hawaii (2007)
24. Gegick, M., Williams, L.: Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs. In: Workshop on Software Engineering for Secure Systems – Building Trustworthy Applications. ACM New York, USA (2005)
25. Howard, J. D., Longstaff, T. A.: A Common Language for Computer Security Incidents. In:

Sandia Report (SAND98-8667). Sandia National Laboratories, California (1998)
26. Tsipenyuk, K., Chess, B., McGraw, G.: Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. In: IEEE Security and Privacy. Volume 3. No. 6. pp. 81--84. (2005)
27. Hansman, S., Hunt, R.: A taxonomy of network and computer attacks. J. Computer and Security. Volume 24, Issue 1, 31 -- 43, Elsevier Science Ltd (2005)
28. Hansmann, S.: A Taxonomy of Network and Computer Attacks Methodologies. In: Technical Report. Department of Computer Science and Software Engineering, University of Canterbury, New Zealand (2003)
29. Killourhy, K. S., Maxion, R. A., Tan, K. M. C.: A Defense-Centric Taxonomy Based on Attack Manifestations. In: International Conference on Dependable Systems and Networks. pp. 91 – 100. IEEE Press, Los Alamitos, CA (2004)
30. Igure, V., Williams, R.: Taxonomies of Attacks and Vulnerabilities in Computer Systems. J. IEEE Communications Surveys and Tutorials. Volume 10, Issue 1. 6 – 19 (2008)
31. Shahriar, H., Zulkernine, M.: Taxonomy and Classification of Automatic Monitoring of Program Security Vulnerability Exploitations. J. Systems and Software 84, 250--269 (2011)
32. Sotirov, A. I.: Automatic Vulnerability Detection Using Static Source Code Analysis. M. Sc. Thesis. University of Alabama (2005)
33. Moore, H. D.: Exploiting Vulnerabilities. In: Secure Application Development (SECAPPDEV). Secappdev.org (2007)
34. Metasploit Penetration Testing Framework, http://www.metasploit.com/framework/modules/
35. Symantec Threat Explorer. http://www.symantec.com/business/security_response/threatexplorer/vulnerabilities.jsp
36. Wagner, D.: Static Analysis and Computer Security: New Techniques for Software Assurance. Phd. Thesis. University of California, Berkeley (2000)
37. Security Development Lifecycle (SDL) Banned Function Calls. http://msdn.microsoft.com/en-us/library/bb288454.aspx
38. Stanford University: Pintos Project. http://www.stanford.edu/class/cs140/projects/pintos/pintos.html#SEC_Top
39. Secunia Advisories. http://secunia.com/advisories/
40. Engler, D.: How to find lots of bugs in real code with system-specific static analysis. http://www.stanford.edu/class/cs343/mc-cs343.pdf
41. Ashcraft, K., Engler, D.: Using Programmer-written Compiler Extensions to Catch Security Holes.IEEE Symposium on Security And Privacy, pp. 143--159 (2002)
42. Red Hat Bugzilla: Bug 546621. https://bugzilla.redhat.com/show_bug.cgi?id=546621
43. National Vulnerability Database: Vulnerability Summary for CVE-2010-4409. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-4409
44. Integer Overflow. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Integer+Overflow
45. CWE-190: Integer Overflow or Wraparound. http://cwe.mitre.org/data/definitions/190.html
46. Richarte, G.: Multiple Vulnerabilities in Stack Smashing Protection Technologies. Security Advisory, Core Labs (2002)
47. Stack Overflow. http://www.owasp.org/index.php/Stack_overflow
48. Lhee, K., Chapin, S. J.: Type-Assisted Dynamic Buffer Overflow Detection. In: 11th USENIX Security Symposium. USENIX Association, CA, USA (2002)
49. Nelißen, J.: Buffer Overflows for Dummies. SANS InfoSec Reading Room - Threats/Vulnerabilities. SANS Institute (2003)
50. Nergal: The Advanced Return-into-lib(c) Exploits. Phrack Magazine. Volume 11, Issue 58, (2001)
51. Using Environment for Returning Into Lib C. http://www.securiteam.com/securityreviews/5HP020A6MG.html
52. Grenier, L. A.: Practical Code Auditing. Metasploit Framework (2002)
53. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing Memory Error Exploits with WIT. In: IEEE Symposium on Security and Privacy. pp. 263--277. (2008)
54. Tevis, J. J., Hamilton, J. A.: Methods for the Prevention, Detection and Removal of Software Security Vulnerabilities. In: 42nd annual Southeast Regional Conference. pp. 197--202. (2004)
55. SecurityFocus. http://www.securityfocus.com/archive/1/515362
56. Microsoft Security Bulletin MS03-029. http://www.microsoft.com/technet/security/bulletin/ms03-029.mspx
57. iDefense Labs Public Advisory: 06.12.07. http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=542
58. CVE-2005-3828. http://www.cvedetails.com/cve/CVE-2005-3848/
59. Testing for Heap Overflow. http://www.owasp.org/index.php/Testing_for_Heap_Overflow
60. Double Free. http://www.owasp.org/index.php/Double_Free
61. CWE-415: Double Free. http://cwe.mitre.org/data/definitions/415.html
62. Kolmonen, L.: Securing Network Software using Static Analysis. In: Seminar on Network Security. Helsinki University of Technology (2007)

63. Nagy, C., Mancoridis, S.: Static Security Analysis Based on Input-related Software Faults. In: European Conference on Software Maintenance and Reengineering. pp. 37--46. IEEE Computer Society (2009)
64. Durden, T.: Automated Vulnerability Auditing in Machine Code. Phrack Magazine. Issue 64 (2007)
65. Michael, C., Lavenhar, S. R.: Source Code Analysis Tools – Overview. Homeland Security (2006)
66. Wagner, D., Foster, J. S., Brewer, E. A., Aiken, A.: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In: Network and Distributed System Security (2000)
67. C Language Issues for Application Security. http://www.informit.com/articles/article.aspx?p=686170&seqNum=6
68. Pozza, D., Sisto, R. : A Lightweight Security Analyzer inside GCC. In: 3rd International Conference on Availability, Reliability and Security. pp. 851--858. Barcelona (2008)
69. Morin, J.: Type Conversion Errors. In: Black Hat. USA (2007)
70. FFmpeg Type Conversion Vulnerability. http://securityreason.com/securityalert/5033
71. CWE-704: Incorrect Type Conversion or Cast. http://cwe.mitre.org/data/definitions/704.html
72. CWE-195: Signed to Unsigned Conversion Error. http://cwe.mitre.org/data/definitions/195.html
73. STR34-C. Cast characters to unsigned char before converting to larger integer sizes. https://www.securecoding.cert.org/confluence/display/seccode/STR34-C.+Cast+characters+to+unsigned+char+before+converting+to+larger+integer+sizes
74. Black, P. E., Kass, M., Kog, M.: Source Code Security Analysis Tool Functional Specification Version 1.0. In: NIST Special Publication 500-268. (2007)
75. Seacord, R. C.: The CERT C Secure Coding Standard. Addison-Wesley Professional (2008)
76. Unintentional Pointer Scaling. http://www.owasp.org/index.php/Unintentional_pointer_scaling
77. Uninitialized variable. http://en.wikipedia.org/wiki/Uninitialized_variable
78. Eight C++ programming mistakes the compiler won't catch. http://www.learncpp.com/cpp-programming/eight-c-programming-mistakes-the-compiler-wont-catch/
79. Uninitialized Variable. http://www.owasp.org/index.php/Uninitialized_Variable
80. Flake, H.: Attacks on Uninitialized Local Variables. Black Hat Federal (2006)
81. CWE-170: Improper Null Termination. http://cwe.mitre.org/data/definitions/170.html
82. Microsoft Security Bulletin MS09-056. http://www.microsoft.com/technet/security/bulletin/ms09-056.mspx
83. CVE-2007-0042. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0042
84. SolarDesigner: Getting around non-executable stack (and fix). Bugtraq Mailing List. http://www.securityfocus.com/archive/1/7480