Deduplication and Exploitability Determination of UAF Vulnerability Samples by Fast Clustering

Jianshan Peng[1], Mi Zhang[2], Qingxian Wang[1]

[1] China National Digital Switching System Engineering and Technological Research Center, ZhengZhou 450002 China

[e-mail: jxpjs@163.com]

[2] Henan Technical College of Construction, ZhengZhou 450002 China

*Corresponding author: Jianshan Peng

Received May 3, 2016; revised July 18, 2016; accepted August 30, 2016; published October 31, 2016

Abstract

Use-After-Free (UAF) is a common lethal form of software vulnerability. By using tools such as Web Browser Fuzzing, a large amount of samples containing UAF vulnerabilities can be generated. To evaluate the threat level of vulnerability or to patch the vulnerabilities, automatic deduplication and exploitability determination should be carried out for these samples. There are some problems existing in current methods, including inadequate pertinence, lack of depth and precision of analysis, high time cost, and low accuracy. In this paper, in terms of key dangling pointer and crash context, we analyze four properties of similar samples of UAF vulnerability, explore the method of extracting and calculate clustering eigenvalues from these samples, perform clustering by fast search and find of density peaks on a large number of vulnerability samples. Samples were divided into different UAF vulnerability categories according to the clustering results, and the exploitability of these UAF vulnerabilities was determined by observing the shape of class cluster. Experimental results showed that the approach was applicable to the deduplication and exploitability determination of a large amount of UAF vulnerability samples, with high accuracy and low performance cost.

Keywords: Use After Free, deduplication of vulnerability samples, exploitability determination, clustering, dangling pointer

1. Introduction

Vulnerability has been a serious threat to the commercial softwares, Web Services[1], Internet of Things[2], and even home automation systems[3]. Use-After-Free (UAF) vulnerability [4] is one type of vulnerabilities that can be commonly found in web browser software, e.g., Internet Explorer, Chrome, and Firefox. The fast and automatic discovery of vulnerabilities can be conducted on the web browser software via fuzzing test tools [5]. Internet Explorer 8, for instance, uses “nduja,” a vulnerability discovery tool for web browser software, for its security testing, and about 100 UAF vulnerability samples can be generated within 24 hours. If a multi-machines parallel test is adopted, the number of vulnerability samples will increase proportionally with the number of testing machines. Due to the characteristics of fuzzing test, most samples are real and repeatable, leading to crashes of the software under test. It is necessary to analyze the causes of theses vulnerability samples, in order to evaluate the threat level of the software vulnerability or to patch these vulnerabilities.

Before analysis, since a variety of samples may be derived from the same vulnerability (we call these samples are in the same vulnerability category), they should be classified and deduplicated. Moreover, the exploitable and unexploitable vulnerabilities respectively represent different threat level, they should be determined through analysis of vulnerability samples. As it is very difficult to manually classify the duplicated samples and to determine the exploitability of a vulnerability among a large number of samples, an automatic approach must be used, in an attempt to achieve the following goals.

Goal 1 To automatically classify the UAF samples, in order to merge the duplicated samples.

A vulnerability is unique means that the root cause of it is different from the other ones. For example, CVE-2014-1776 vulnerability is caused by mistakenly releasing the “CMarkup” object [6], while CVE-2013-3893 vulnerability is caused by mistakenly releasing the “CTreeNode” object [7]. It is impossible to classify samples according to the sample contents, because there may be great differences between the contents of UAF samples although they are in the same category. For example, to trigger a web browser software vulnerability, a large number of javascript codes will be generated and the key one is submerged in a lot of redundant ones. Unfortunately, the vast majority of them are not associated. In this paper, according to the root cause of the samples, we propose a novel approach to fast screening of duplicated samples by clustering.

Goal 2 To automatically determine the exploitability of UAF vulnerability.

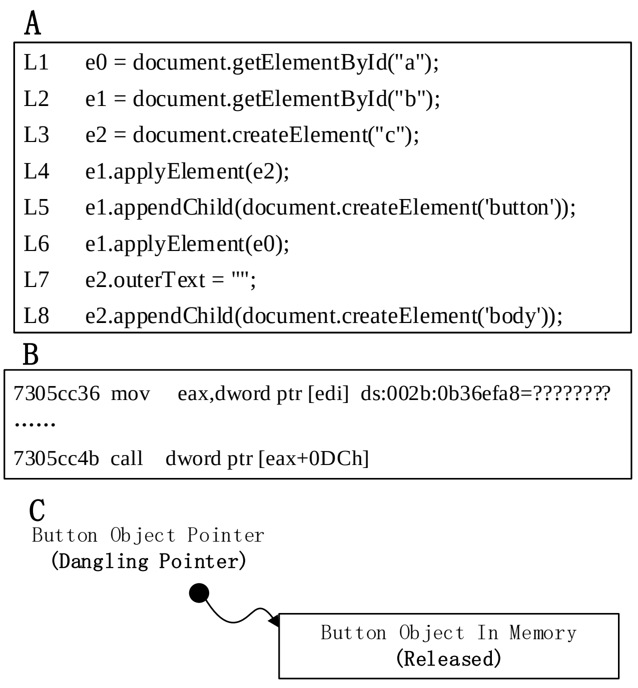
Being exploitable means that one vulnerability can not only destroy but also tamper the execution flow of the program, enabling execution of arbitrary instructions. In this paper, we propose an idea to automatically determine the exploitability of UAF vulnerability based on the characteristics of exploiting UAF vulnerability.

The remaining of the paper is organized as follows: In Section 2, the classification principles and exploitability determination principles, as well as the difficulties in the implementation process, of UAF vulnerability were discussed and defined. In Section 3, four properties of UAF vulnerability were investigated, the clustering approach was selected to achieve the goals, and a detailed description of the implementation process was provide. In Section 4, the feasibility of the approach was demonstrated by experimental data, with low calculation cost and extremely high accuracy.

2. Problem Overview

2.1 Related Definition

UAF vulnerability refers to the condition where an object is created and then released in later runtime, but the pointer that points to this object still remains in the memory. In the subsequent process of the program, the program will crash if this pointer was used to access the memory area of this released object. UAF vulnerability is common in web browser software. Taking CVE-2012-4782 vulnerability [8] as an example, the critical javascript codes of the vulnerability sample is shown in **Fig. 1A**.



A

```
L1    e0 = document.getElementById("a");
L2    e1 = document.getElementById("b");
L3    e2 = document.createElement("c");
L4    e1.applyElement(e2);
L5    e1.appendChild(document.createElement('button'));
L6    e1.applyElement(e0);
L7    e2.outerText = "";
L8    e2.appendChild(document.createElement('body'));
```

B

```
7305cc36  mov    eax,dword ptr [edi]  ds:002b:0b36efa8=????????
……
7305cc4b  call   dword ptr [eax+0DCh]
```

Fig. 1. CVE-2012-4782 sample. (**A**) The critical javascript code.
(**B**) The x86 instructions causing crash. (**C**) The schematic diagram of dangling pointer.

Three objects of element, i.e., *e0*, *e1*, and *e2*, were created in line L1-L3, respectively. Their corresponding parent-child relationship was created in L4-L6 in the sequence of *e2->e0->e1->"button,"* All child elements of *e2* were eliminated in L7, including the newly created "*button*" object, then the reuse of "*button*" object was caused in L8. But at this time, since the memory occupied by "*button*" object had been released, Use-After-Free vulnerability was triggered.

The x86 instructions causing crash is shown in **Fig. 1B**, where *0x0b36efa8*, the value of *edi* register, was the pointer of "*button*" object, pointing to a memory area which had been released. As a result, access violation exception occurs in the program. In this case, the pointer of "*button*" object was a dangling pointer [9], as shown in **Fig. 1C**.

The dangling pointer was defined as follows.

Definition 1 The pointer variable *p* is a dangling pointer, if and only if

$$\left(x := allocate(size) \mid 0 \le x \le 2^{32}-1\right) \wedge \left(p \in [x, x+size-1]\right) \wedge \left(release(x)\right)$$

denoted dangling pointer as p_{DP}. It was based on the assumption that the program was running on the x86 instruction system.

In summary, UAF vulnerability is defined as follows.

Definition 2 A program crash is UAF vulnerability, if and only if the dangling pointer is generated and reused during the execution process of the program. The formalization is defined as follows:

$$Crash \in \{Vul_{UAF}\} \Leftrightarrow (Crash \leftarrow (Access(p_{DP})))$$

It should be noted that not all the dangling pointers generated by program will cause crash, only the **key dangling pointer** (noted **KDP**) which cause crash is concerned. The KDP should be taken as the basis to determine whether the multiple UAF samples belong to the same vulnerability category. For instance, in **Fig. 1A**, the KDP is "*button*," which is the basis for determining the CVE-2012-4782 vulnerability category. The principle for classifying the vulnerability categories is defined as follows.

Definition 3 Several UAF samples belong to the same category of vulnerability, if and only if the KDP in them is generated from the same piece of program codes which creating (i.e., allocating) and releasing it.

Similarly, in terms of patching vulnerabilities, if the program codes generating the KDP can be located and fixed, then all the samples in this vulnerability category will be invalid and this vulnerability thus will be fundamentally patched.

These are the conditions that generate and trigger the UAF vulnerabilities, as well as the principle for classifying their categories. To exploit UAF vulnerability, "memory occupying" should be carried out, that is, before the dangling pointer is reused, the pointed memory area is first filled as the controllable data. Still, taking CVE-2012-4782 as example, the "memory occupying" codes are shown in **Fig. 2**.

```
for(var i = 0; i<0x150; i++) {
    occur_arr[i] = document.createElement("div");
    occur_arr[i].title = junk.substring(0,(0x58-6)/2);
}
```

Fig. 2. The "memory occupying" codes for CVE-2012-4782

A total of *0x150* "*div*" objects were created in this piece of codes, and the size of each object was carefully calculated to be exactly 0x58 bytes, the size of "*button*" object, and then the memory of "*button*" object which had been just released was refilled. When the vulnerability was triggered, *edi* register in **Fig. 1B** would point to the content of "*div*" object. Since the data in this memory was controllable, *eax* register was controlled, and then the execution flow was tampered by the subsequent instruction *call[eax+0xdc]*. Arbitrary code can be executed by combining the HeapSpray technique [10].

Definition 4 A UAF vulnerability is exploitable, if and only if pointed memory area can be occupied by controllable data, before the KDP is reused. The formalization is defined as follows.

$$Crash \in \{Vul_{UAF}\} \cap \{Vul_{exploitable}\} \Leftrightarrow$$

$$\left(\left([p_{DP}+l] \leftarrow data_{controlled} \mid l \in [0, sizeof(p_{DP})]\right)_{t_1} \wedge \left(Access(p_{DP})\right)_{t_2} \mid t_1 < t_2\right)$$

where t_1 and t_2 respectively represent the time where events occur.

There may be a case where the memory area cannot be occupied, and the C codes in **Fig. 3** illustrates this case.

```
L1    char *p = (char*)malloc(100);
L2    free(p);
L3    *p = 'a';
```

Fig. 3. The unexploitable UAF vulnerability

Obviously, crash would be caused by L3. But this vulnerability is unexploitable, as between L2 and L3, there is no opportunities to occupy the memory area pointed by pointer *p* by using the controllable data.

2.2 Difficulty Analysis

To achieve Goal 1 and Goal 2, there are two main difficulties as follows.

Difficulty 1 To achieve Goal 1, we first need to analyze why the KDP was generated in samples, then classify some samples as the same vulnerability category if they are caused by the same KDP. It should be noted that there is diversity in the way the same UAF vulnerability generated crash, which was depending on when and where the KDP was reused.

For instance, due to the blindness of Fuzzing test, the memory pointed by the KDP in the vulnerability sample might be occupied by various objects, such as “*div*” or “*image*,” before being reused. This would cause the program crash in different time and instructions, e.g., when invoking property or method of “*div*” or “*image*,” Therefore, the difficulty in classifying the UAF vulnerability samples is about how to identify their most fundamental causes from a variety of program crash scenes, i.e., find and compare the KDP of them. Based on the method of dynamic taint analysis [11], Undangle [12] tracks the allocations and releases of all pointers in the program, and is effective in finding and eliminating the dangling pointers. However, this tool must run on TEMU simulator [13]. Limited to its running speed, Undangle will take up to 1982 seconds to analyze a complex vulnerability sample. For this reason, this method is not suitable for analyzing a large number of vulnerability samples.

Difficulty 2 To achieve Goal 2, we need to determine whether there is an opportunity to occupy the memory pointed by the KDP with controllable data before the KDP was reused. An analysis of the internal processes of the program would be necessary, and the situation could be very complex. For web browser software, “having the opportunity to occupy” means that, after a HTML or javascript object is released, and before it is reused, there is opportunity to execute javascript codes to occupy the memory, which often occurs between two statements of javascript; for network communication software, “having the opportunity to occupy” means that before the KDP is reused, the program can reads network data and allocate memory for these data. So far, there has been no automatic method of determining exploitability in UAF vulnerability, while the determination method by manually analyzing the program process is not only time consuming but also inaccurate.

3. Method and Implement

By debugging and observing a large number of UAF samples, we found that similar samples showed measurable patterns and similarities in two dimensions--the creation and release of the KDP, and the context where crash occur. Moreover, there was significant difference between the exploitable and unexploitable vulnerability categories. Therefore, in this paper, on the basis of the fast clustering algorithm proposed by Alex Rodriguez et al. [14], an approach was put forward to fast and accurately classify different UAF samples and to determine the exploitability of UAF vulnerability, thereby achieving Goals 1 and 2.

3.1 Observation of Samples

We can’t determine vulnerability category according to the content of samples, however, the samples those in the same vulnerability category share similarities on the running track in the program. According to observation on a lot of samples, UAF vulnerabilities have the following four properties.

Property 1 The information of the same KDP are restored in the crash context of the samples in the same vulnerability category.

When the program crash are caused by the KDP, the context where crash occur will surely restore some information of this pointer, for instance, a register may be pointing to this KDP, such as *edi* register shown in **Fig. 1B**. Otherwise, a register may be in the range of this pointer, or else, a *dword* value in the current stack is associated with it. This property can be used to quickly determine the range of the KDP among various pointers.

Property 2 The running tracks of the samples in the same vulnerability category contain the same creation and release codes of the KDP.

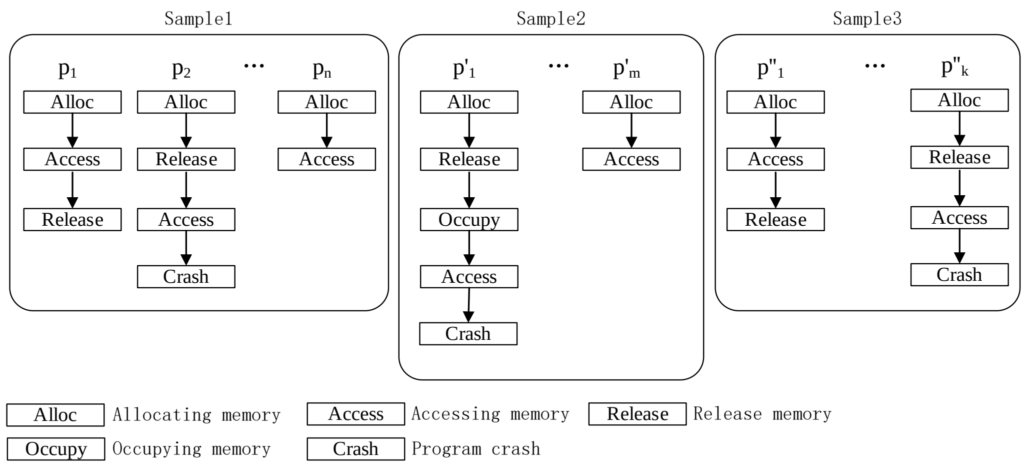


Fig. 4. The use of pointers in separately samples

Fig. 4 illustrates this property, here *p* represents the memory pointers in the program. If both *sample1* and *sample2* belong to the same vulnerability category, then p_2 and $p_1^{'}$ will have the similar *Alloc* and *Release* blocks. If *sample2* and *sample3* do not belong to the same vulnerability category, then $p_1^{'}$ and $p_k^{''}$ will show significant difference in their *Alloc* and *Release* blocks. Hence this property can be used as one of the bases for measuring clustering similarities.

It should be noted that due to the diversity of the samples, *Access* block and *Occupy* block may show great differences in samples even if they belong to the same category, so we don’t take *Access* block and *Occupy* block as measurement basis of clustering.

Property 3 There are similarities in the crash context of the samples in the same vulnerability category.

Deriving from the same vulnerability, the similar samples show identical or similar context when causing crash in program. In **Fig. 4**, if *sample1* and *sample3* belong to the same vulnerability category, it indicates that they are both triggered by the same KDP(p_2 and

$p_k^{''}$). Since the memory is not occupied by other objects (i.e., there is no *Occupy* block for p_2 and $p_k^{''}$), the location of crash instructions is exactly the same, and their *Crash* blocks are also exactly the same. On the contrary, if they do not belong to the same vulnerabilities category, their *Crash* blocks are completely different.

Furthermore, we assume that *sample1* and *sample2* belong to the same vulnerability category, they are both triggered by the same KDP(p_2 and $p_1^{'}$). Since there is one more *Occupy* block in the KDP($p_1^{'}$) of *sample2*, leading to their completely different but associated *Crash* blocks. We can illustrates it in **Fig. 1B**. If there is no *Occupy* block, and the memory pointed by *edi* register has been released and thus cannot be accessed, then *mov* instruction will be crash. Otherwise if there is *Occupy* block, and the memory pointed by *edi* has been occupied by other objects, then *mov* instruction is normal, but the subsequent *call* instruction may be crash, because the memory pointed by *eax* register may be not accessible. It should be noted that the locations of *mov* and *call* instructions are close to each other. Therefore, this property can also be used as one of the bases for measuring clustering similarities.

Property 4 The exploitable and unexploitable UAF vulnerabilities are significantly different in the composition of categories.

If the exploitable UAF vulnerabilities have been tested for long enough, among the obtained abundant sample sets, there are inevitably some samples that can occupy the memory pointed by the KDP, and some samples that cannot. Therefore the exploitable UAF vulnerabilities simultaneously have two subcategories of samples, which are relevant but significant different. The relevance between them apparently originates from properties 1 and 3, while the significant difference refers to their crash context—the samples that can occupy the memory will crash in diverse locations, while others generally crash in the same location. We have explained this situation in the example in Property 3. We also can deduce that the unexploitable UAF vulnerabilities only have one subcategory of samples, those do not have opportunities to occupy the memory and always crash in the same location. Therefore, the exploitable and unexploitable UAF vulnerabilities are significantly different in the composition of categories, and this property can be used to determine the exploitability of a vulnerability.

3.2 Determination of Methods and Algorithms

Based on the above analysis, it can be found that the similar samples are similar in two dimensions: 1) The creation and release codes of the KDP, and 2) the crash context. In this regard, the eigenvalues of a lot of UAF samples were extracted and calculated, mapped as points in a two-dimensional diagram. Then the clustering algorithm was adopted to complete the sample clustering to obtain the classification graph, in which the samples in the same class cluster were regarded as the same vulnerability category, thereby achieving the deduplication of the samples. Moreover, the exploitability of one vulnerability was determined according to Property 4.

The clustering algorithm adopted in this paper was proposed by Alex Rodriguez et al. in 2014 [14]. This algorithm is short and efficient, assuming that the center of class cluster is surrounded by neighbor points with low local density and is relatively far from any points with higher density. For each data point *i*, two values should be calculated: its local density, ρ_i, and its distance δ_i from points of higher density, both of which depend on the distance d_{ij} between data points.

The local density ρ_i of data point *i* is defined as $\rho_i = \sum_j \chi\left(d_{ij} - d_c\right)$, where $\chi(x)$ function is defined as $\chi(x) = \begin{cases} 1, x < 0 \\ 0, x \geq 0 \end{cases}$, and d_c represents a cutoff distance. In fact, ρ_i is equal to the number of the points with a distance less than d_c to point *i*. δ_i of data point *i* is the minimum distance from point *i* to any point with higher density, defined as $\delta_i = \min_{j:\rho_j > \rho_i}\left(d_{ij}\right)$. For the point with the maximum density, $\delta_i = \max_j\left(d_{ij}\right)$ is set. The points with a great ρ_i and a great δ_i are considered to be the center of the class cluster, after that all the other points belong to the clusters represented by the nearest cluster centers.

This algorithm can identify the class clusters with various shapes, and the clustering is achieved without iteration. It has been proven to have lower calculation cost, faster speed, and better clustering than the conventional clustering algorithm, such as K-means [15].

3.3 Record of Pointers Information

By debugging each UAF sample, the breakpoints were set in the API function of allocating memory and releasing memory. For example, *rtlAllocateHeap* and *rtlFreeHeap* functions in *ntdll.dll* in the Windows platform. When the above functions were invoked, the related information of the pointers were automatically recorded, including:

1) The call stack when the allocating API was invoked, denoted as set $\{alloc\_stack_i \mid i \in [0, n_{alloc}]\}$.

2) The pointer value returned after invoking an allocating API, denoted as set $\{alloc\_ptr_i \mid i \in [0, n_{alloc}]\}$.

3) The size of the allocated memory in bytes, denoted as set $\{alloc\_size_i \mid i \in [0, n_{alloc}]\}$.

4) The call stack when the releasing API was invoked, denoted as set $\{release\_stack_i \mid i \in [0, n_{release}]\}$.

5) The pointer value when memory release, denoted as set $\{release\_ptr_i \mid i \in [0, n_{release}]\}$.

3.4 Selection of Candidate Sets of the KDP

When program crash, the crash instructions were analyzed, and the current call stack was denoted as *crash_stack*, before the following operations:

1) To calculate *crash_addr*, which is the memory address causes "Access Violation," if

$$\left(crash\_addr \in [alloc\_ptr_i, alloc\_ptr_i + alloc\_size_i - 1] \mid i \in [0, n_{alloc}]\right) \wedge$$

$$\left(\exists j \in [0, n_{release}], release\_ptr_j = alloc\_ptr_i\right)$$

suggesting that $alloc\_ptr_i$ is directly related to the KDP, and thus should be given a high weighted value. All points *i* meeting the requirement are denoted as candidate set $I_{highweight}$.

2) To scan 8 values of general registers, denoted as $\{reg\_addr_k \mid k \in [1,8]\}$, if

$$\left(reg\_addr_k \in [alloc\_ptr_i, alloc\_ptr_i + alloc\_size_i - 1] \mid k \in [1,8], i \in [0, n_{alloc}]\right)$$

$$\wedge \left(\exists j \in [0, n_{release}], release\_ptr_j = alloc\_ptr_i\right)$$

suggesting that $alloc\_ptr_i$ is not directly related to the KDP, and thus should be given a medium weighted value. All points *i* meeting the requirement are denoted as candidate set $I_{midweight}$.

3) To scan all values of *dword* in the current stack, denoted as $\{instack\_addr_k \mid k \in [0, n_{instack}]\}$, if

$$\left(instack\_addr_k \in [alloc\_ptr_i, alloc\_ptr_i + alloc\_size_i - 1] \mid k \in [0, n_{instack}], i \in [0, n_{alloc}]\right)$$

$$\wedge \left(\exists j \in [0, n_{release}], release\_ptr_j = alloc\_ptr_i\right)$$

suggesting that is indirectly related to the KDP, and thus should be given a low weighted value. All points *i* meeting the requirement are denoted as candidate set $I_{lowweight}$.

The purpose of this step is as follows: The information of thousands to hundreds of thousands of points will be recorded by step 3.3, where only one or several pointers are related to the KDP. According to Definition 1, the dangling pointer must be a released one, and according to Property 1, the crash context will restore the information of the KDP. Three candidate sets of KDP with different weighted values can be selected from a lot of pointers by the above 3 requirements, $I_{highweight}$, $I_{midweight}$ and $I_{lowweight}$, in order to facilitate the next extraction and calculation of the eigenvalues.

3.5 Quantification of Call Stack Eigenvalue

The call stack information includes the instruction address, function name, function parameters, etc., and is not suitable for calculating the eigenvalue. It should be quantified. The principle of quantification is to extract and normalize the most representative address in the call stack. The calculation method is to take each line of instruction address in the call stack into the form of (DLL name + address offset value). The ASCII code values of the DLL names are added, while the hexadecimal value of the address offset value are added every 8 bits, then the weighted sum of the two are calculated. The formula is as follows.

$$(\sum_{i=1}^{lengh(DLLname)} byte\_in\_DLLname[i]) \times 0x100 + (\text{address\_offset} \ \& \ 0xff) + ((\text{address\_offset} >> 8) \ \& \ 0xff)$$
$$+((\text{address\_offset} >> 16) \ \& \ 0xff) + ((\text{address\_offset} >> 24) \ \& \ 0xff)$$

The final result is a 32-bit value similar to the CRC checksum. For example, if the call stack is:

ntdll+0x00064142
ntdll+0x00007760
kernel32+0x00152340

Then the calculated result is:

*(0x6E+0x74+0x64+0x6C+0x6C)*0x100+（0x00+0x06+0x41+0x42）*

*+（0x6E+0x74+0x64+0x6c+0x6c）*0x100+（0x00+0x00+0x77+0x60）*

*+（0x6B+0x65+0x72+0x6E+0x65+0x6C+0x33+0x32）*0x100+（0x00+0x15+0x23+0x40) =* ***0x723D8***

Instead of directly calculating the instruction address, the form of (DLL name + address offset value) is adopted to avoid being affected by ASLR technique [16] in operating system. Moreover, instead of Hash algorithm, a weighted sum is adopted to ensure that a similar calculated result can be obtained from the similar instruction address, to facilitate the calculation of similarity.

3.6 Extraction and Calculation of Clustering Eigenvalue

Two clustering eigenvalues of each sample were extracted and calculated, respectively:
1) The eigenvalues of the KDP. The call stack in the 3 candidate sets of the KDP was quantified to obtain the eigenvalue by using the method in Section 3.5, and the corresponding eigenvalue sets were respectively denoted as $alloc\_value_i \mid i \in I_{highweight}$, $alloc\_value_i \mid i \in I_{midweight}$ and $alloc\_value_i \mid i \in I_{lowweight}$, which were then added by the following formula:

$$\sum_{i \in I_{highweight}} (alloc\_value_i \times \frac{0.5}{|I_{highweight}|}) + \sum_{j \in I_{midweight}} (alloc\_value_j \times \frac{0.3}{|I_{midweight}|}) +$$
$$\sum_{k \in I_{lowweight}} (alloc\_value_k \times \frac{0.2}{|I_{lowweight}|})$$

With the weighted proportion of 0.5, 0.3, and 0.2, this formula normalized the eigenvalue of 3 candidate sets to the range of 32 bits.

2) The crash context eigenvalue. The call stack of crash instruction, *crash_stack*, was quantified to obtain eigenvalue *crash_value*, by using the method in Section 3.5, and the eigenvalue was also normalized to the range of 32 bits.

3.7 Clustering

The crash context eigenvalues were mapped to the two-dimensional *x*-axis, the eigenvalues of the KDP were mapped to the y-axis. The clustering algorithm introduced Section 3.2 was used to perform clustering analysis of these data points, to obtain the decision graph. The samples corresponding to the data points belonging to the same category of cluster were classified as the same vulnerability category. This clustering algorithm requires to manually designate a cutoff distance parameter d_c. In addition, to automatically select the center point of class cluster, the points with δ_i and ρ_i within the appropriate range should be selected. The known classified data were used for training to ultimately obtain the appropriate range of δ_i and ρ_i. The clustering process is detailed in the experimental analysis in Section 4.

3.8 Determination of Exploitability

We According to Property 4, the exploitable and unexploitable vulnerabilities are significantly different in the clustering results. The latter only include the sample clusters do not perform *Occupy,* so there is no significant difference in *x* values between the data points. However, the former include both the sample clusters perform and do not perform *Occupy*, and there is no significant difference in *y*-axis between the data points of these two class clusters, but there is significant difference in *x*-axis (but it is not significant enough to affect the clustering results). Hence, by observing the shape of the class cluster, the exploitability of the vulnerabilities can be determined. See Section 4.5 for details.

3.9 System Design

Based on the methods and analysis above, we implemented the ADEDU(Automatic Deduplication and Exploitability Determination of UAF) system. **Fig. 5** illustrates the overview of this system.

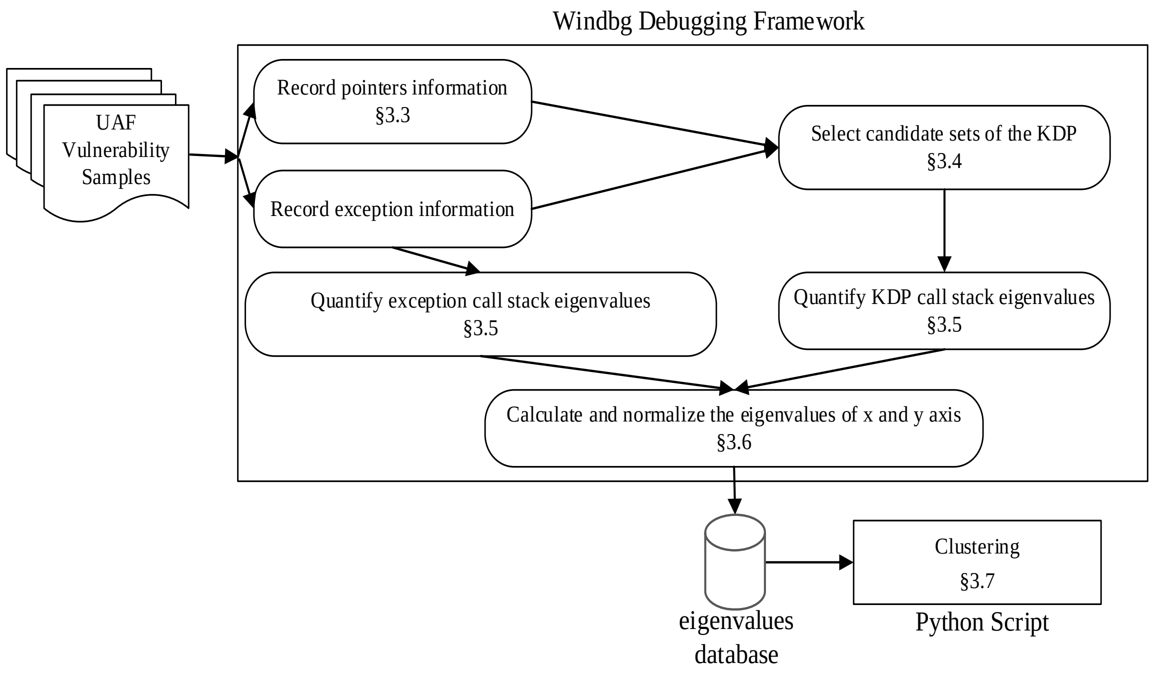

Windbg Debugging Framework
UAF Vulnerability Samples
Record pointers information §3.3
Record exception information
Select candidate sets of the KDP §3.4
Quantify exception call stack eigenvalues §3.5
Quantify KDP call stack eigenvalues §3.5
Calculate and normalize the eigenvalues of x and y axis §3.6
eigenvalues database
Clustering §3.7
Python Script

Fig. 5. Overview of ADEDU's design.

Firstly, UAF vulnerability sample is delivered to the windbg debugger tool in order to debug it's process, then we can obtain the x and y axis data which are necessary for clustering. We programed a C plugin for windbg to provide the functionalities descripted in Section 3.3, 3.4, 3.5 and 3.6. The x-axis data are obtained from follow way: 1) Recording pointers information and exception information. 2) Selecting candidate sets of the KDP by the method descripted in Section 3.4. 3) Quantifying call stack represented by candidate sets. 4) Normalizing the eigenvalue of call stack. Similarly, the y-axis data are obtained from call stack too. However, it is not represented by candidate sets, but by exception context when program crash.

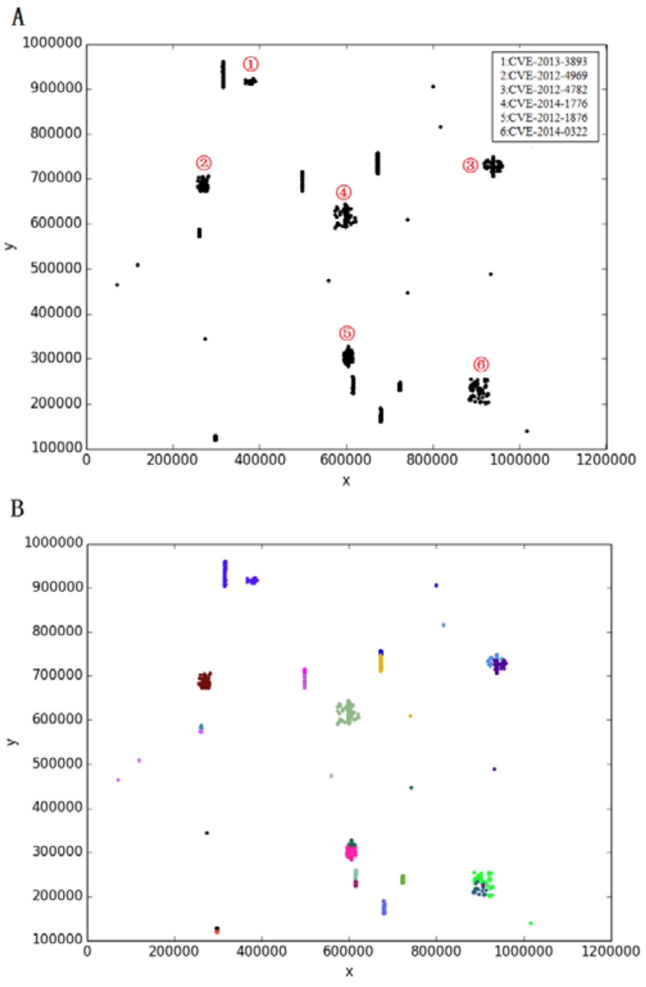
Secondly, we use these data for clustering, which is descripted in Section 3.7, although the process of tuning parameters is detailed in Section 4.

4. Experimental Studies

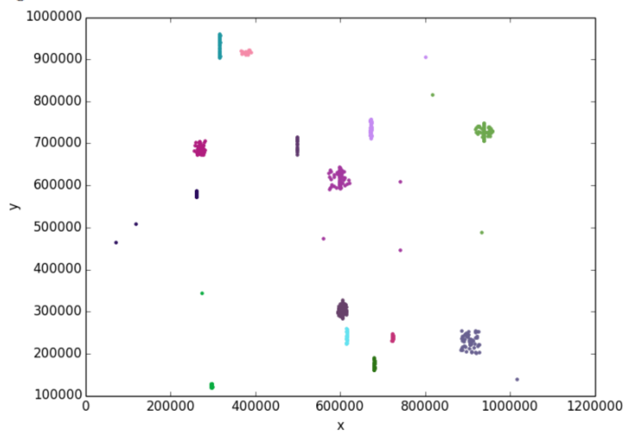
In this study, the hardware platform was the Intel i7 4GHz processor and 16GB memory. The software platform was the 64-bit version of Windows 7 SP1 operating system. To obtain a lot of UAF vulnerability samples, we improved "nduja" tool by adding the new test objects, attributes and methods, and used this tool simultaneously on 10 computers to test the 32-bit version of Internet Explorer(version 8.0.7600.16385) for 3 consecutive days, thus obtaining 1,236 vulnerability samples. Meanwhile, 6 samples of known exploitable vulnerabilities in

Internet Explorer were collected to verify the correctness and effectiveness of the proposed approach, so there are 1,242 samples in total.

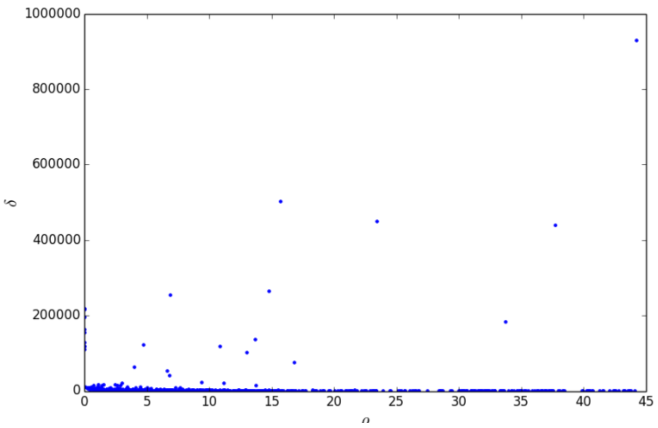
We divided 1,242 samples into 2 sets randomly--1,041 samples(including 6 samples of known exploitable vulnerabilities) and 201 samples. It took a total of 12,692 seconds by using ADEDU to extract and calculate the clustering eigenvalues of the 1,041 samples, with an average of 12.2 seconds for each sample. **Fig. 6A** shows the distribution of these samples after their eigenvalue were calculated, with x-axis representing the crash context eigenvalues, and y-axis representing the KDP eigenvalues. ①~⑥ in the figure represent the class clusters where the samples of 6 known vulnerabilities existed. After 8 seconds of short-term calculation, the final clustering results was obtained as shown in **Fig. 6C**, where different colors represent different class clusters. They were classified into 14 class clusters, 6 of them contained the samples of known vulnerabilities, and the remaining 8 ones were the new class clusters. We also clustered the remaining 201 samples to verify the correctness of parameters. The clustering process and results were further analyzed in the following part.



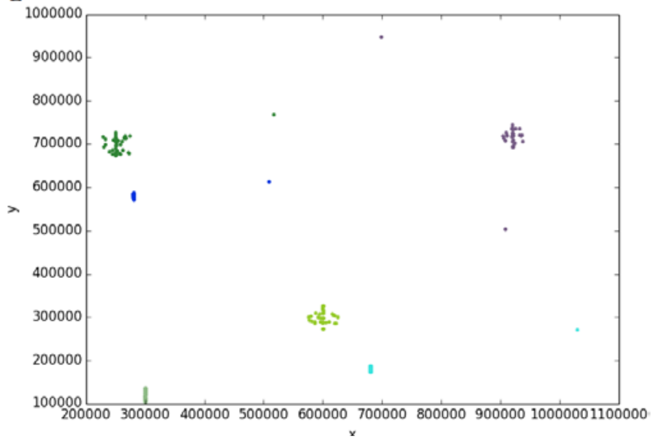
C


1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
y
0
200000
400000
600000
800000
1000000
1200000
x

D


1000000
800000
600000
400000
200000
0
δ
0
5
10
15
20
25
30
35
40
45
ρ

E


1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
y
200000
300000
400000
500000
600000
700000
800000
900000
1000000
1100000
x

F

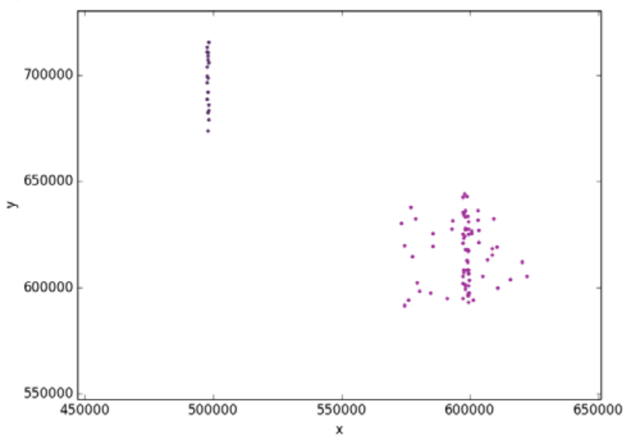


Fig. 6. Clustering process. (**A**)Pointer distribution of 1041 samples. (**B**)Clustering results with Cut-Off-Kernel algorithm. (**C**)Clustering results with Gaussian-Kernel algorithm. (**D**)Decision graph for (C). (**E**)Clustering result of 201 samples. (**F**)Partially enlarged view of (C).

4.1 Selection of Cutoff Distance Parameter d_c

Reference [15] points out that the algorithm is only sensitive to the relative value of ρ_i. Hence for the large data sets, the analysis result is considerably robust to the selection of the cutoff distance parameter d_c. In this paper, the Euclidean distances of all points to the other ones were sorted from small to large, and the x-th value was selected as d_c. Study showed that a x selected from 2% to 10% would not affect the final clustering results.

4.2 Improvement of Local Density Algorithm

Reference [15] uses the Cut-Off-Kernel algorithm to calculate the local density ρ_i. Result shows that this algorithm is very effective in clustering the spherical class cluster, but not for the aspherical ones. However, in **Fig .6A**, most class clusters were the aspherical ones. The clustering effect of the Cut-Off-Kernel algorithm is shown in **Fig. 6B**, where several class clusters were mistakenly classified into 2 or more clusters, indicating that the Cut-Off-Kernel algorithm could not properly handle the data in this paper. Therefore, the Gaussian-Kernel algorithm [17] was adopted, with the calculation formula as follows.

$$\rho_i = \sum_j \left(e^{-\left(\frac{d_{ij}}{d_c}\right)^2}\right)$$

The clustering effects were shown in **Fig. 6C**, and the different class clusters were accurately classified.

4.3 Automatic Selection of Class Cluster Center

Fig. 6D shows the calculated values of ρ_i and δ_i. As suggested by the figure, the points with great values of ρ_i and δ_i are characterized by multiple neighboring points and far distance from the center points of other class cluster. They should be considered as the center points of the class clusters. However, to automatically select these points, a threshold value should be set for ρ_i and δ_i. The calculation method in this paper is as follows

$$\rho_{thres} = rate_\rho \times (\max(\rho_i) - \min(\rho_i)) + \min(\rho_i)\,,\quad \delta_{thres} = rate_\delta \times (\max(\delta_i) - \min(\delta_i)) + \min(\delta_i)$$

After many experiments, we set $rate_\rho$=0.07, $rate_\delta$=0.05. To verify the correctness of them, we clustered 201 samples by using the same value of $rate_\rho$ and $rate_\delta$. As **Fig. 6E** shows, the clustering results is satisfactory.

4.4 Elimination of Noise

There are several isolated points in **Fig. 6C**, which may be due to one of the following three conditions.

1) It belong to a vulnerability category, but it is significantly different from the other samples in the category.

2) It belong to some vulnerability category, which contains only one sample.

3) It belong to a not-UAF vulnerability category, and the points around y=0 are mostly of this type. Since the causes of not-UAF vulnerability are irrelevant with the dangling pointers, it is difficult to find the candidate set of the KDP, leading to a clustering eigenvalue of 0 on y-axis.

All these isolated points were referred to as noise in this paper, which had no significant impacts on the experimental results after being eliminated. The elimination method was to find the points with small ρ_i but large δ_i from the decision graph in **Fig. 6D**. This feature suggested that these points had few neighbors and were very far from the other center points, should be set as noises. The calculation method is omitted.

4.5 Difference Between Exploitable and Unexploitable UAF Sample Clusters

Fig. 6F is the partially enlarged view of **Fig. 6C**. The left class cluster presents the class cluster of unexploitable vulnerabilities, while the right presents the exploitable ones. It can be obviously seen that both of them show a great changes in y values, because there are difference between the KDP candidate sets of samples. It can be also obviously seen that the left class cluster shows little change in x values and are distributed in a strip, because the crash context of them are unitary, while the right one shows great changes in x values and are scattered, because the released memory may be occupied by various objects, leading to multiple crash locations in the program. Moreover, the right class cluster obviously contain both the subcategories that perform and do not perform *Occupy*, which is consistent with Property 4. It is observed that there are 6 exploitable vulnerability categories and 8 unexploitable vulnerability categories in **Fig. 6C**.

4.6 Performance Comparison

The parameter d_c was adjusted, and the Gaussian-Kernel algorithm was employed to replace the Cut-Off-Kernel algorithm. The threshold values of ρ_i and δ_i were set to automatically select the center of cluster, and the decision diagram was used to eliminate the noises. Through this series of processes, the 1,041 samples were classified into 14 different categories, with the statistical results in **Fig. 7.** In the figure, 6 were determined as exploitable vulnerability samples, which was consistent with the known vulnerabilities, and 8 were determined as unexploitable vulnerability samples, which was manually verified. Another 15 noises were mistakenly classified. Through manual verification, 4 of them were not-UAF samples, 8 were new unclassified samples, and 3 belonged to the other classified categories. The primary cause of noise is the too small number of similar samples. The accuracy will be further improved, if more similar samples can be obtained through longer test.

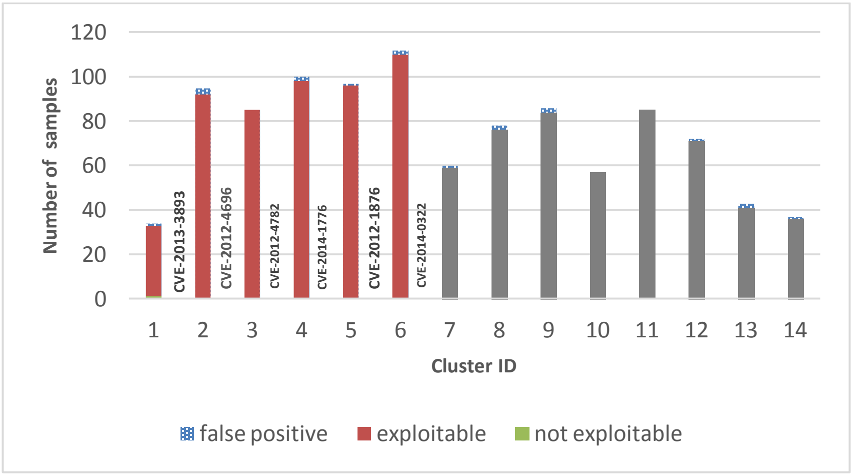


Fig. 7. Cluster result statistics**.**

We also compared the performance of ADEDU with similar tools as described below.

A. Runtime Comparison

Table 1 illustrates the comparison results of runtime between ADEDU, FileFuzz[22], MSEC[27] and UnDangle[12] which can find dangling pointers. ADEDU analyzed 1041 samples in 12692 seconds, average 12.2 seconds per sample. UnDangle tool must run on the TEMU simulator to carry out taint analysis, at a slow running speed. As it takes 165 seconds to analyze a IE8 sample, this method is not suitable to analyze a large number of vulnerability samples. Although FileFuzz and MSEC run faster than ADEDU benefit from their simple analysis mechanisms, they get very low accuracy in deduplication and exploitability determination, as showed in Part B and C.

Table 1. Runtime Comparison Result

	Object	Base Platform	Analysis Mechanisms	Runtime (Seconds)
ADEDU	IE8	Windbg	Recording pointers, call stack, using clustering algorithm.	12.2 (Average)
UnDangle	IE8	TEMU	Taints Analysis.	165
FileFuzz	IE8	None	Compare call stack of exception.	6.4 (Average)
MSEC	IE8	Windbg	Analyzing context of exception.	7.8 (Average)

B. Comparison of Accuracy in Samples Deduplication

Table 2 illustrates the comparison result of accuracy in samples deduplication between ADEDU and FileFuzz, which is used to find vulnerabilities and deduplicate samples. The method used by FileFuzz is to summarize the crash context, and thus to compare the summaries so as to determine whether they are the same vulnerability. As this method only focuses on the limited information, it cannot distinguish the samples of the same vulnerability cause in different forms of crash, and its accuracy in deduplicating UAF samples is very low.

Table 2. Comparison Result of Accuracy in Samples Deduplication

	Samples Number	Number of Category	Number of Correctly Classified Samples	Number of Wrongly Classified Samples	Accuracy Rate
ADEDU	1041	14	1026	15	98.6%
FileFuzz	1041	856	167	874	16%

C. Comparison of Accuracy in Exploitability Determination

Table 3 illustrates the comparison result of accuracy in exploitability determination between ADEDU and MSEC, which can help to determine the exploitability when a program crashes during debugging. In ADEDU, since deduplication and exploitability determination are completed in the same clustering process , the accuracy rates of them are same, i.e. 98.6%. On the contrary, MSEC can only provide static analysis of the stack environment and code context during crashes, lacking the dynamic analysis of the subsequent data flow and control flow, with a low accuracy.

Table 3. Comparison Result of Accuracy in Exploitability Determination

	Samples Number	Number of Exploitable Samples		Number of Unexploitable Samples		Accuracy Rate
		Correct	Wrong	Correct	Wrong	
ADEDU	1041	517	3	509	12	98.6%
MSEC	1041	327	193	231	290	53.6%

5. Related Work

Studies on UAF vulnerability: Some debugging tools such as Purify [18] can find the dangling pointers by checking whether the pointers are pointing to the live memory. But this method cannot detect the dangling pointers for which the memory areas have been reused. Electric Fence tool[19] and PageHeap technique [20] use a new page for each allocation and rely on page protection mechanisms to detect dangling pointer uses, but it can hardly locate the root of the dangling pointer. TIE [21] uses the constraint solving approach to infer the dangling pointers, with high calculation cost. Based on dynamic taint analysis, UnDangle tool [12] follows the allocation and release of all pointers in the program, and it is evidently effective in finding and eliminating the dangling pointers in UAF vulnerability. However, this tool must run on the TEMU simulator to carry out taint analysis, at a slow running speed. As it takes up to 1,982 seconds to analyze a single complex vulnerability sample, this method is not suitable to analyze a large number of vulnerability samples. In this paper, the candidate set of the KDP was extracted from a large number of pointers, and the eigenvalues can be extracted in a short period of time.

Automatic deduplication of vulnerability samples: The method used by tools such as FileFuzz [22] and SAGE [23] is to summarize the crash context, and thus to compare the summaries so as to determine whether they are the same vulnerability. As this method only focuses on the limited information, it cannot distinguish the samples of the same vulnerability cause in different forms of crash, and its accuracy in deduplicating UAF samples is very low. In this paper, the eigenvalues were extracted from the samples, fast clustering was conducted on a large number of samples, and the samples were accurately deduplicated in a short time according to the clustering results.

Automatic determination of the exploitability of vulnerability: Based on patch analysis, APEG [24] is an automatic generation tool of attack codes that can exploit the vulnerability, requiring the support of the program patch. Following the approach by APEG that the vulnerability exploitability can be described as the predicate of a program's state space, AEG [25] can automatically finish the whole process from vulnerability discovery to exploitability analysis to the generation of attack codes. However, this tool is designed for the source code, and can only handle two types of vulnerabilities, the returned-address overwriting in stack and the format string overflow. MAYHEM [26] integrates the advantages of on-line and off-line symbolic executions and is mainly used for testing binary programs, where determining the exploitability of vulnerability samples is based on modification of AEG. However, lacking the support of type and structural information of high-level languages, its efficiency and accuracy of determination is lower than AEG. In addition, on the basis of Windbg, Microsoft has developed the MSEC tool [27], which can help to determine the exploitability when a program crashes during debugging. This tool can provide static analysis of the stack environment and code context during crashes, lacking the dynamic analysis of the subsequent data flow and control flow, with a very low accuracy. Since the method of exploiting UAF vulnerabilities is unique, the above techniques are not entirely suitable for determining the exploitability of UAF vulnerability. In this paper, the characteristics and properties of UAF vulnerabilities were analyzed, and their exploitability was accurately determined by observing the shapes of class clusters.

6. Conclusions

In this study, by using the fast clustering algorithm by density peaks, an approach was proposed to deduplicate UAF samples and to determine their exploitability. By analyzing the properties of UAF vulnerabilities, an extraction method for the sample eigenvalues was constructed, consuming an average of 12.2 seconds for each sample. An advanced fast clustering algorithm was used to cluster the vulnerability categories of a lot of samples, thereby accurately and rapidly screening duplicated samples, conducting the clustering of a thousand samples in 8 seconds. By observing the shape of class cluster in the clustering results, the exploitability of the UAF vulnerability can be accurately determined, with an accuracy of 98.6%. The proposed approach is particularly suitable for testing the web browser software, where a lot of vulnerability samples are obtained by Web Browser Fuzzing. But there are also some problems, for instance, the accuracy depends on whether the sample set is sufficient, and too few vulnerability samples in a single category will lead to failure to identify this type of vulnerability. In addition to increasing test time to increase the number of samples, the better solutions should be provided in further researches.

References

[1] G. Gugnani, SP Ghrera, et al., “Implementing DNA Encryption Technique in Web Services to Embed Confidentiality in Cloud,” in *Proc. of International Conference on Computer and Communication Technologies*, pp. 407-415, 2015. Article (CrossRef Link).

[2] YE Ning, Y. Zhu, RC Wang, et al., “An Efficient Authentication and Access Control Scheme for Perception Layer of Internet of Things,” *Applied Mathematics & Information Sciences*, vol. 8, no. 4, pp. 1617-1624, 2014. Article (CrossRef Link).

[3] AC Jose, R. Malekian, “Smart Home Automation Security: A Literature Review,” *Smartcr*, vol. 5, no. 4, pp. 269-285, 2015. Article (CrossRef Link).

[4] J. Afek, A. Sharabani, "Dangling Pointer – Smashing the Pointer for Fun and Profit," in *Proc. of BlackHat*, Las Vegas, CA, July 2007.Article (CrossRef Link).

[5] Ruderman, J., "JavaScript fuzzer available," *Mozilla Security Blog*, Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.Article (CrossRef Link).

[6] "CVE-2014-1776 (IE 0day) Analysis," *SIGNALSEC*, Available: http://www.signalsec.com/blog/cve-2014-1776-ie-0day-analysis/, May 1st, 2014. Article (CrossRef Link).

[7] "Microsoft Internet Explorer CVE-2013-3893 Memory Corruption Vulnerability," *SecurityFocus*, Available: http://www.securityfocus.com/bid/62453, Jul 15, 2015.Article (CrossRef Link).

[8] "Microsoft Internet Explorer CMarkup Function Use-After-Free Arbitrary Code Execution Vulnerability," *Cisco Security*, Available: https://tools.cisco.com/security/center/viewAlert.x?alertId=27533, 2012. Article (CrossRef Link).

[9] J. Feist, L. Mounier, ML. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology & Hacking Techniques*, 2014, vol. 10, no. 3, pp. 211-217, 2014. Article (CrossRef Link).

[10] A. Sotirov, M. Dowd. "Bypassing browser memory protections," in *Proc. of Blackhat*, 2008. Article (CrossRef Link).

[11] James Newsome, Dawn Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of Network and Distributed System Security Symposium*, San Diego, CA, USA, 2005. Article (CrossRef Link).

[12] J. Caballero, G. Grieco, M. Marron, A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proc. of the 2012 International Symposium on Software Testing and Analysis ACM*, pp. 133-143, 2012. Article (CrossRef Link).

[13] TEMU: The BitBlaze Dynamic Analysis Component. Article (CrossRef Link).

[14] R. Alex, L. Alessandro, "Clustering by fast search and find of density peaks," *Science*, vol. 344, no. 6191, pp. 1492-6, 2014. Article (CrossRef Link).

[15] J. A. Hartigan, M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Applied Statistics*, vol. 28, no. 1, pp. 100-108, 1979. Article (CrossRef Link).

[16] "Windows ISV Software Security Defenses," *MSDN*, Available: https://msdn.microsoft.com/en-us/library/bb430720.aspx, December 2010. Article (CrossRef Link).

[17] J. Babaud, A. P. Witkin, M. Baudin, et al., "Uniqueness of the gaussian kernel for scale-space filtering," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, vol. 8, no. 1, pp. 26-33, 1986.Article (CrossRef Link).

[18] R. Hastings, B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," in *Proc. of USENIX 1992*, San Francisco, California, 1992.Article (CrossRef Link).

[19] B. Perens. "Electric Fence Malloc Debugger," July 2011.Article (CrossRef Link).

[20] "Page Heap for Chromium," July 2011.Article (CrossRef Link).

[21] J. Lee, T. Avgerinos, D. Brumley, "TIE: Principled Reverse Engineering of Types in Binary Programs," in *Proc. of Network and Distributed System Security Symposium*, San Diego, California, February 2011.Article (CrossRef Link).

[22] S. Sparks, S. Embleton, R. Cunningham, et al., "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting," in *Proc. of Computer Security Applications Conference*, pp. 477-486, 2007.Article (CrossRef Link).

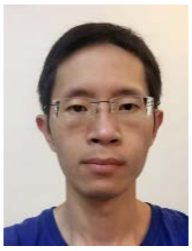
[23] P. Godefroid, M. Y. Levin, D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Proc. of Network and Distributed System Security Symposium*, pp. 151-166, San Diego, CA, 2008. Article (CrossRef Link).

[24] D. Brumley, P. Poosankam et al., "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proc. of 2013 IEEE Symposium on Security and Privacy*, pp. 143-157, 2013. Article (CrossRef Link).

[25] T. Avgerinos, S. K. Cha, A. Rebert, et al., "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74-84, 2014. Article (CrossRef Link).

[26] S. K. Cha, T. Avgerinos, A. Rebert, et al., "Unleashing mayhem on binary code," in *Proc. of IEEE Security and Privacy*, pp. 380-394, San Francisco, 2012. Article (CrossRef Link).

[27] "!exploitable Crash Analyzer - MSEC Debugger Extensions," Available: http://msecdbg.codeplex.com/, 2013. Article (CrossRef Link).



Jianshan Peng received the M.A. degree in Computer Science and Technology from China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, and Ph.D. Candidate in Computer Science and Technology. From 2006 to 2015, he served as an university lecturer. He is currently an associate professor at China National Digital Switching System Engineering and Technological Research Center. His research interests include Cyberspace Security and Software Vulnerability.

Mi Zhang received the M.A. degree in Computer Science and Technology from Zhengzhou University, Zhengzhou, China. She is currently a lecturer in Henan Technical College of Construction, Zhengzhou, China. Her research interest include Cyberspace Security and Computer Security.

Qingxian Wang is a professor and doctoral tutor at China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China. His research interests include Cyberspace Security and Trusted Software.