www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 4 April, 2013 Page No. 1111-1116

A Secure Environment for Unsafe Component Loading

Veeralakshmi S, Sindhuja M

Department of Information Technology, Rajalakshmi Engineering College, Chennai.

Department of Information Technology, Rajalakshmi Engineering College, Chennai.

Veera34@gmail.com

ABSTRACT

Dynamic loading is widely used in designing and implementing software. Its benefits include modularity and generic interfaces for third-party software such as plug-ins. Dynamic loading components are utilization requires local file system access on the end host. The following problems are occurred in the local and remote dynamic components loading. In local system, the file does not exist in the specified Path or the specified search directories, hijacking the components. Although in the remote system, the browser automatically download arbitrary files to the user's Desktop directory without any prompting, vulnerable program starts up via the shortcut, an archive file containing a document and a malicious component. In existing system the admin have to analyze the profile to check unsafe components. The proposed system has a facility to construct a profile for unsafe component by user.

Key terms: *Unsafe component loading, dynamic analysis, Component resolution, Shared object, User Account Control*

1. INTRODUCTION

Dynamic loading is an important mechanism for software development. It allows an application the flexibility to dynamically link a component and use its exported functionalities. Its benefits include modularity and generic interfaces for third-party software such as plug-ins. It also helps to isolate software bugs as bug fixes of a shared library can be incorporated easily. Because of these advantages, dynamic loading is widely used in designing and implementing software. A key step in dynamic loading is component resolution, i.e., locating the correct component for use at runtime. Operating systems generally provide two resolution methods, either specifying the full path or the filename of the target component.

With full path, operating systems simply locate the target from the given full path. With filename, operating systems resolve the target by searching a sequence of directories, determined by the runtime directory search order, to find the first occurrence of the component. Although flexible, this common component resolution strategy has an inherent security problem. Since only a file name is given, unintended or even malicious files with the same file name can be resolved instead. Thus far this issue has not been adequately addressed. In particular, it shows that unsafe component loading represents a common in class of security vulnerabilities on the Windows and Linux platforms. Software components often utilize functionalities exported by other components such as shared libraries at runtime. This operation is generally composed of three phases: resolution, loading, and usage. Specifically, an application resolves the needed target components, loads them, and utilizes the desired functions provided by them. Component interoperation can be achieved through dynamic loading provided by operating systems or runtime environments. For example, the Load Library and dlopen system calls are used for dynamic loading on Microsoft Windows and Unix-like operating systems, respectively. Dynamic loading is generally done in based on component resolution and chained component loading. It have discovered new remote attack vectors based on the findings from analysis, which Microsoft confirmed and actively worked with us and other software vendors to develop engineering solutions to patch. The project also discusses and proposes techniques to mitigate unsafe component loadings.

Although dynamic loading is a critical step in software execution, it also has an inherent security implication. Specifically, a loaded target component is only determined by the specified file name. This can lead to the loading of unintended or even malicious components and

thus may allow arbitrary code execution. For example, an attacker can trick a vulnerable web browser to resolve a spyware file with the specified file name instead of the intended component. Some of the world's most popular Windows programs are vulnerable to attacks that exploit a major bug in the way they load critical code libraries, according to sites tracking attack code. Automatic detection describes security vulnerabilities and threats in dynamic component loading, including remote code execution attacks based on unsafe dynamic loadings. It represents general technique for detecting unsafe dynamic loadings.

2. RELATED WORKS

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney[3] proposed a tool for automated detection for unsafe component Pin is dynamic binary instrumentation framework for the IA-32 and x86-64 instruction set architectures that enables the creation of dynamic program analysis tools, As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code. Pin was originally created as a tool for computer architecture analysis, but its flexible API and an active community have created a diverse set of tools for security, emulation and parallel program analysis.

Pin tool is platform independence. C. Grier, S. Tang, and S.T. King[2] suggested op web browser for secure environment. Current web browsers provide attackers with easy access to modern computer systems. A new web browser that is designed to support web-based applications securely, called the OP web browser. One difficulty in analyzing browser-based attacks is that the activities of the attacker are intermingled with legitimate actions. OP web browser used to overcome these attacks to enable users and system administrators to better understand browser-based attacks.

Fuzz testing is an effective technique for finding security vulnerabilities in software. Godefroid, M.Y. Levin, and D.A. Molnar[5] proposed fuzz testing tools apply random mutations to well-formed inputs of a program and test the resulting values. Fuzz testing is a form of black box random testing which randomly mutates well-formed inputs and tests the program on the resulting data.

P. Saxena, P. Poosankam, S. McCamant, and D. Song[10] proposed Loop- Extended Symbolic Execution on Binary Programs to reduce bugs. Mixed concrete and symbolic execution is an important technique for finding and understanding software bugs, including security relevant ones. The symbolic execution technique is loop-extended symbolic execution, which generalizes from a concrete execution to a set of program executions which may contain a different number of iterations for each loop as in the original execution. This tool finds vulnerabilities in both standard benchmark suite and real world application. I. Goldberg, D. Wagner, R. Thomas, and E.A. Brewer[6] implemented a secure Environment for Untrusted Helper Applications Confining the Wily Hacker.

Netscape use untrusted helper applications to process data from the network. The aim is to confine the untrusted software and data by monitoring and restricting the system calls it performs. Web browsers are increasing popular tool for retrieving data form network. Helper applications also apply to web browser. Helper applications should not be able to communicate with outside network. KLEE is a program analysis tool that works by symbolic execution and constraint solving, finding possible inputs that will cause a program to crash, and outputting these as test cases. C. Cadar, D. Dunbar, and D. Engler[1] support klee is a capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally intensive programs. KLEE, automatically generated tests that, on average, covered over 90% of the lines (in aggregate over 80%) in roughly 160 complex, system-intensive applications ”out of the box.” Klee generated tests that achieve high line coverage. Dynamic testing is a term used in software engineering to describe the testing of the dynamic behavior of code. Jacob Burnim and Koushik Sen[7] proposed Heuristics for Scalable Dynamic Test Generation. That is, dynamic analysis refers to the examination of the physical response from the system to variables that are not constant and change with time. In dynamic testing the software must actually be compiled and run.

In random testing, the program under test is simply executed on randomly-generated inputs. It can effectively test large programs. D. Molnar, X.C. Li, and D.A. Wagner[5] implemented dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs Integer bugs have been increasing sharply and become the notorious source of bugs for various serious attacks. In this tool, IntFinder, this can automatically detect Integer bugs in an x86 binary program. We implement IntFinder based on a combination of static and dynamic analysis. Dynamic test generation is better suited to finding such bugs, and we develop new methods for finding a broad class of integer bugs with this approach. We have implemented these methods in a new tool, Smart Fuzz that analyzes traces from commodity Linux x 86 programs. D.Brumley, D.X. Song, T. Chiueh, R. Johnson, and H. Lin[4] proposed Automatically Protecting against Integer-Based Vulnerabilities in Network and Distributed System.

RICH(Run-time Integer C Hecking), a tool for efficiently detecting integer-based attacks against C programs at run time. The RICH compiler extension compiles C programs to object code that monitors its own execution to detect integer-based attacks. This paper surveys integer based attacks and provides a theoretical framework to formally define and reason about integer errors soundly. O. Ruwase and M.S. Lam[8] proposed A Practical Dynamic Buffer Overflow Detector is most common form of security threat in software systems and vulnerabilities attributed to buffer overflows have consistently dominated C Range Error Detector advisories. CRED proved effective in detecting buffer overrun attacks on programs with known vulnerabilities, and is the only tool found to guard against a test bed of 20 different buffer overflow attacks.

3. PROPOSED SYSTEM

In existing system the admin have to analyze the profile to check unsafe components. The proposed system has a facility to construct a profile for unsafe component by user.

3.1 Architecture

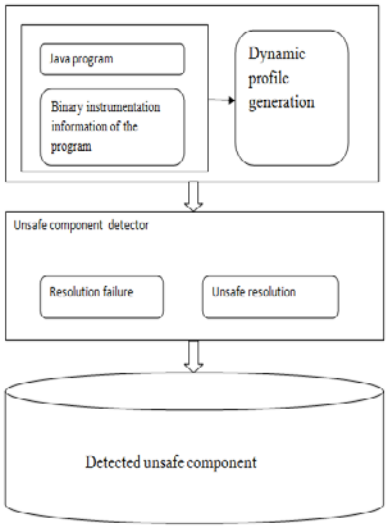

Figure 3.1 Proposed architecture diagram.

Figure 4.1 represents the architecture diagram of the system. It is two three tier architecture consists of binary instrumentation and unsafe component detector generate the database and it has been connected with user. In client Dynamic binary instrumentation checks the behavior of the input program. Unsafe component detector checks the resolution failure and unsafe resolution. Database is an repository which stores all the details of the malicious program.

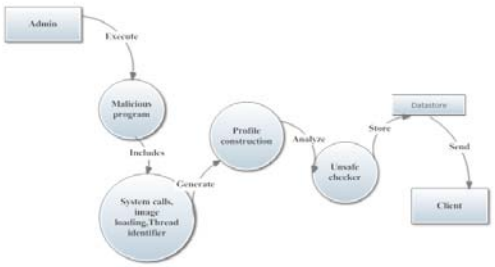

Figure 3.2 Proposed DFD diagram

Figure 4.1 represents the overall DFD for the proposed system. User gives the malicious program as s input and its run by admin. Admin generate the profile based on system call, image loading and thread identifier. This profile will forward to the user.

3.2 Malicious program

Malicious program is software used or created by attackers to disrupt computer operation, gather sensitive information, or gain access to private computer systems. It can appear in the form of code, scripts, active content, and other software. Malware is a general term used to refer to a variety of forms of hostile or intrusive software Malware is not the same as defective software, which is software that has a legitimate purpose but contains harmful bugs that were not corrected before release. A java program creates with error and bugs and malicious program files. This program is executed by the admin.

3.3 Profile construction

Constructs the profile from the running program. This profile contain following three information such as system calls, image loading, Thread process and identifiers. System calls invoked for dynamic loading for information on target component specifications, directory search orders, and the sequence of component loading behavior. Capture actual loadings of target components via dynamic binary instrumentation. The loading information is needed for reconstructing the loading procedure in a combination with the information captured by the system call instrumentation. It also indicates the resolved full path determined by the loading procedure. The target program uses multithreads and each thread loads a component dynamically, the instrumented system calls for each loading can be

interleaved, which makes it difficult to correctly reconstruct the loading procedure of each thread.

3.4 Unsafe checker

To detect unsafe component resolutions first capture a sequence of system-level actions for dynamic loading during a program's execution. Use dynamic binary instrumentation to generate the profile on its runtime execution. Then reconstruct the dynamic loading information from the profile offline and check safety conditions for each resolution. Because our technique only requires binary executables, it is robust and can be applied to analyze not only open source applications but also commercial off-the-shelf products.

3.5 Identify the result

Detected unsafe components results are identify by the user. This component contains information about malicious code.

4. IMPLEMENTATION

4.1 Directory search order

Dynamic component resolution based on filename requires a directory search order, which is determined by system and program settings at runtime. According to MSDN the SafeDllSearchMode registry key, the LOAD_WITH_ ALTERED_SEARCH_PATH flag, and the SetDllDirectory system call determine five possible types of directory search orders at runtime, which are standard search order (Safe- DllSearchMode), alternate search order (SafeDllSearchMode), and SetDllDire-ctory-based SearchOrder.

Type	Conditions
Resolution failure	1. Target component is not found
Unsafe resolution	1. Target component is specified by its name 2. Target component is resolved by iterating through multiple directories 3. There exists another searched directory before the resolution

Figure 5.1 Conditions for Detecting Unsafe Component Loadings

4.2 Chained DLL loading

According to Microsoft, there exist two types of load-time dependencies among DLLs: implicit dependency and forwarded dependency.

4.2.1 Implicit dependency: If a DLL A and a DLL B are linked at compile/link time, and the source code of DLL A calls one or more functions exported from DLL B, DLL A has implicit dependency on DLL B. Note that implicit-dependent DLLs are determined by function calls invoked by the source code of the loading DLL. Even though the function is not invoked at runtime, the DLL exporting the function is also loaded.

4.2.2 Forwarded dependency: While this dependency is similar to the implicit dependency, it differs in what the DLL that implements the invoked functions is. For the load-time dependency, the functions that a loading DLL invokes are directly implemented in its dependent DLLs. However, for forwarded dependency, the implementation of the invoked function call simply forwards control to the actual code implemented in another DLL. In this case, the loading DLL has forwarded dependency on the DLL containing the forwarded implementation.

```
1 (dd0,98c) LdrLoadDll IMESHARE.DLL
2             C:\Program Files\Microsoft Office\Office14;
3             C:\Windows\system32;C:\Windows\system;
4             C:\Windows;.;$PATH
5 (dd0,98c) LdrpApplyFileNameRedirection IMESHARE.DLL
6             NOT_REDIRECTED
7 (dd0,98c) LdrpFindLoadedDllByName IMESHARE.DLL NOT_LOADED
8 (dd0,98c) LdrpFindKnownDll IMESHARE.DLL UNKNOWN
9 (dd0,98c) LdrpSearchPath IMESHARE.DLL FAILED
```

Fig 5.2 A resolution failure in Microsoft Word 2010.

Based on this information stored in the profiles, we perform offline analysis to detect unsafe DLL loadings.

5. EVALUATION

Evaluate unsafe component loadings on Microsoft Windows and Linux. For each platform, detect unsafe component loadings in a diverse selection of popular applications.

Table 1: Number of Detected Unsafe DLL Loadings

Software	Windows XP					Windows Vista					Windows 7				
	Failed			Unsafe		Failed			Unsafe		Failed			Unsafe	
	Fullpath	Filename		Filename		Fullpath	Filename		Filename		Fullpath	Filename		Filename	
	T	T	C	T	C	T	T	C	T	C	T	T	C	T	C
MS Office															
Excel 2010	0	1	0	10	14	0	1	0	6	8	0	2	0	14	9
OneNote 2010	0	0	0	7	12	0	1	0	26	21	0	0	0	34	8
Outlook 2010	2	2	0	27	24	2	2	0	22	20	2	1	0	15	23
PowerPoint 2010	1	2	0	16	20	1	2	0	16	16	1	2	0	19	13
Publisher 2010	2	1	0	17	20	2	2	0	11	23	2	1	0	18	16
Word 2010	2	2	0	10	17	1	1	0	16	11	15	2	0	18	10
Sub total	7	8	0	87	107	6	9	0	97	99	20	8	0	118	79
Web Browser															
Chrome 6.0.472.63	1	1	0	22	17	1	1	0	9	8	1	1	0	30	13
Firefox 3.6.10	1	1	0	18	5	1	1	0	19	14	1	0	1	16	10
IE 8.0 / 9.0 Beta	0	0	0	20	17	0	0	0	17	15	0	0	0	21	6
Opera 10.63	2	1	0	9	8	0	1	0	6	22	0	1	0	14	9
Safari 5	0	1	0	13	59	0	0	0	9	36	0	0	0	19	30
Sub total	4	4	0	82	106	2	3	0	60	95	2	2	1	100	68
PDF Reader															
Acrobat Reader 9.4.0	0	0	0	6	9	0	0	0	17	11	0	0	0	5	5
Foxit Reader 4.2	0	0	0	14	10	0	1	0	20	12	0	0	0	9	17
Sub total	0	0	0	20	19	0	1	0	37	23	0	0	0	14	22
Messenger															
Google Talk Beta	0	1	0	21	10	0	0	0	14	8	0	1	0	28	19
Pidgin 2.7.3	0	0	2	11	30	0	0	2	10	33	0	0	2	13	35

Table 1 shows the number of unsafe DLL loadings detected from a few different types of major applications on Microsoft Windows family. In particular, we classify detected failed and unsafe resolutions in terms of the specification type (i.e., fullpath or filename) and the phase at which the unsafe loadings happen. The columns labeled T and C correspond to target and chained component loadings, respectively. Note that the C column is missing for fullpath. This is because components for the chained loading are specified by their filenames. According to the table, unsafe DLL loadings are common programming mistakes in developing these applications. We found more than 3,200 instances of unsafe dynamic loadings: 1,072 under XP, 1,080 under Vista, and 1,117 under Windows 7. Considering the types of these unsafe DLL loadings, unsafe resolution is responsible for almost all of them.

5.1. Performance

To evaluate the performance of our technique, we measure the execution time of each phase for analyzing MS Office products on Windows 7 running on a Core2 Duo 2.40 GHz

Table 2: Execution Time for Analyzing MS Office 2010

Software	Failed			Unsafe	
	Fullpath	Filename		Filename	
	T	T	C	T	C
Email Client					
Balsa 2.4.1	0	0	0	0	0
Evolution 2.28.3	1	1	0	4	155
Kmail 1.13.2	0	36	0	2	10
Thunderbird 3.0.8	0	0	0	0	67
Sub total	1	37	0	6	232
Web Browser					
Chrome 6.0.472.63	0	0	0	0	0
Firefox 3.6.10	1	0	0	5	103
Konqueror 4.4.2	0	39	0	4	8
Opera 10.62.6438	0	1	0	0	0
Seamonkey 2.0.8	0	0	0	3	113
Sub total	1	40	0	12	224
PDF Reader					
Acrobat Reader 9.3.4	0	0	0	5	70
Foxit Reader 1.1	0	0	0	0	0
Sub total	0	0	0	5	70

Table 3 shows the execution time for the profile generation and analysis phases of the analyzed applications. In the evaluation, use default documents as inputs to the analyzed programs. Our results show that our technique is practical and can be effectively applied for analyzing real-world programs such as MS Office.

Table 3: Number of Detected Unsafe SO Loadings

Software	Generation (s)	Analysis (ms)
Excel 2010	51	31
OneNote 2010	36	13
Outlook 2010	116	35
PowerPoint 2010	61	23
Publisher 2010	51	26
Word 2010	83	28

6. CONCLUSION AND FUTURE WORK

The system has been designed for to identify and eliminate the unsafe dynamic component loading in the system. More over the system has been designed for providing security from downloading malicious program. The analysis on the requirements and a design for the proposed system has been screened. The requirement analysis process includes learning and determining about the working environment, technical requirements and logical aspects or features of the system.

To evaluate the technique implemented tools to detect unsafe component loadings on Microsoft Windows and Linux.

7. ACKNOWLEDGEMENT

I would like to thank Mrs.M.Sindhuja, Assistant Professor (SS), Information Technology, Rajalakshmi Engineering College, for guiding me in writing this paper.

REFERENCES

[1] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Proc. Eighth USENIX Conf. Operating Systems Design and Implementation, pp. 209-224, 2008.

[2] C. Grier, S. Tang, and S.T. King, "Secure Web Browsing with the OP Web Browser," Proc. IEEE Symp. Security and Privacy, pp. 402416, 2008.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, "Pin: Building Customized Program Analysis

Tools with Dynamic Instrumentation,"Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, pp. 190-200, 2005.

[4] D. Brumley, D.X. Song, T. Chiueh, R. Johnson, and H. Lin, "RICH: Automatically Protecting against Integer-Based Vulnerabilities,"Proc. Network and Distributed System Security Symp, Mar. 2007.

[5] D. Molnar, X.C. Li, and D.A. Wagner, "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs," Proc. 18th Conf. USENIX Security Symp, pp. 67-82, 2009.

[6] I. Goldberg, D. Wagner, R. Thomas, and E.A. Brewer, "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker," Proc. Sixth Conf. USENIX Security Symp. Focusing on Applications of Cryptography, 1996.

[7] Jacob Burnim and Koushik Sen," Heuristics for Scalable Dynamic Test Generation" Automated Software Engineering, 2008. ASE 2008.

[8] O. Ruwase and M.S. Lam, "A Practical Dynamic Buffer Overflow Detector," Proc. Network and Distributed System Security Symp., Feb. 2004

[9] P. Godefroid, M.Y. Levin, and D.A. Molnar, "Automated White box Fuzz Testing," Proc. Network and Distributed System Security Symp. Mar2008.

[10] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop- Extended Symbolic Execution on Binary Programs," Proc. 18th Int'l Symp. Software Testing and Analysis.

.