Detecting and Preventing ActiveX API-Misuse Vulnerabilities in Internet Explorer*

Ting Dai, Sai Sathyanarayan, Roland H.C. Yap, and Zhenkai Liang

School of Computing
National University of Singapore
{daiting,sathya,ryap,liangzk}@comp.nus.edu.sg

Abstract. ActiveX is used to build reusable software components in Microsoft Windows. It is widely used by many Windows applications, such as Internet Explorer and Microsoft Office. As general-purpose components, ActiveX controls expose methods to applications, which may be used in ways unexpected by the ActiveX designer, leading to malicious activities. We call such misuse of ActiveX methods – *ActiveX API misuse vulnerabilities*. In this paper, we present a solution which identifies and prevents API misuse of ActiveX controls in Internet Explorer. We construct models to represent normal functionality of ActiveX methods, and identify ActiveX API misuse by identifying the methods that can reach dangerous (system) APIs. We then develop a technique for Internet Explorer to prevent the use of dangerous ActiveX methods. We evaluated our approach on six real-world ActiveX controls. We are able to identify and prevent ActiveX API misuse in these controls. Our approach is effective in detecting ActiveX API misuse and has negligible overhead for preventing attacks.

1 Introduction

ActiveX is Microsoft technology to build reusable software components on the Microsoft Windows platform. It is widely used by many Windows applications, including Microsoft Office, Windows Media Player, and Internet Explorer (IE), allowing the applications to use the functionality embedded in ActiveX controls. IE allows methods in ActiveX controls to be accessed from web pages. ActiveX controls in IE are native binaries running with the same privilege of the IE process, thus giving web pages the ability to run native code in the operating system.[1]

Since ActiveX controls are general-purpose components, they often contain more functionality than what is needed by the applications using them. The methods of an ActiveX control may be used in unintended ways. For example, the Snapshot Viewer ActiveX control (installed with Microsoft Office) can be leveraged by malicious web pages in IE to create or overwrite files. Due to the complexity of the Windows system, even if users are aware of the functionality of such ActiveX controls, they may not be able to foresee how the functionality is used. We call this class of vulnerabilities *API*

* This work has been supported by a DRTech grant R-394-000-054-232.

[1] The Windows Update ActiveX Control in Windows XP and Visual Studio (Windows 7) uses IE to apply Microsoft updates.

T.W. Chim and T.H. Yuen (Eds.): ICICS 2012, LNCS 7618, pp. 373–380, 2012.

```
<script language='JavaScript'>
//Create Activex Object with ProgID
var obj = new ActiveXObject("snpvw.Snapshot Viewer Control.1");
// invoke method SnapshotPath, CompressedPath, ...
obj.SnapshotPath = "c:\\TestSnapshot.snp";
obj.CompressedPath = "c:\\TestSnapshot-compressed.snp";
obj.PrintSnapshot("True");
</script>
```

Fig. 1. A JavaScript code snippet using Microsoft Office Snapshot Viewer ActiveX Control in IE

misuse: an API of a software component, such as ActiveX, is misused by programs in ways unexpected by the API designer. We remark that this class of attacks is caused by misuse of an API rather than a case of misused user authority as in the confused-deputy problem.

Existing ActiveX security mechanisms are insufficient to prevent API-misuse attacks. IE's ActiveX security is based on trust. IE trusts ActiveX controls installed locally in the Windows system, except those blocked by compatibility flags in the registry (*killbits*). For remote ActiveX controls, the user provides a white list of trusted sites and permit remote ActiveX control from the white list. For untrusted ActiveX controls, IE asks the user for permission to use the control. In addition, IE implicitly trusts the control as it only initializes and utilizes the ActiveX interfaces where the *Safe for Initialization* and *Safe for Scripting* properties are implemented by the control. Once vulnerability in an ActiveX control is known, the typical solution is to completely block it but that means that all the functionality performed by the control in IE is lost.

In this paper, we develop a solution to identify and prevent API-misuse vulnerabilities in ActiveX controls. Our approach consists of an offline detection phase and an online prevention phase with proxy-based filtering. In the detection phase, we represent the normal functionality of the ActiveX control in a graph model which gives the reachability of ActiveX methods during program execution. This model is generated through dynamic analysis on standard test cases. We then identify API misuse by analyzing the access paths in the model which lead to dangerous APIs. In the prevention phase, we create a proxy to intercept every ActiveX method invocation in IE and block dangerous ActiveX methods. We are able to identify six real-world documented API-misuse vulnerabilities, of which, three are from Microsoft. Our prevention mechanism has low overheads and blocks dangerous methods while preserving other functionality compared with existing solutions.

1.1 A Motivating Example

We show the API misuse problem with a real-world example. The Microsoft Office Snapshot Viewer ActiveX Control is a component of Microsoft Office Access 2003. It is locally installed by default. Thus, is trusted by IE to generate print previews of Office documents.

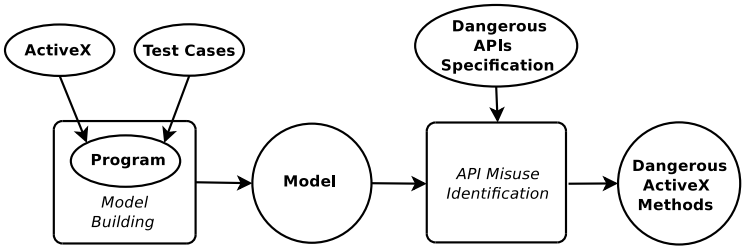

Fig. 2. Overview of our approach on API-misuse detection

The `SnapshotPath` and `CompressedPath` methods of the ActiveX are to specify the path of the snapshot file to be displayed in the Snapshot Viewer. Fig. 1 shows a typical usage of the control. It renders a print preview of the snapshot file `TestSnapshot.snp` and also saves the compressed version of the file. This ActiveX control can be exploited to allow an attacker to download remote files to the local file system or overwrite local files. The attacker invokes `SnapshotPath` with `http://malicious.com/payload.exe` to silently download the payload and invokes `CompressedPath` with `c:\\clickme.exe` to save the malicious payload to the local file system.

Such API misuse vulnerabilities are accomplished through the normal functionality in ActiveX controls. Unlike malware, this type of vulnerability is hard to detect because the program behavior is not malicious by itself. Instead, it is a "misuse" of standard or inherent functionality because of the complexity of the operating systems. Hence, we need a systematic approach to detect API misuse vulnerabilities in ActiveX controls.

2 API-Misuse Detection and Prevention

In this section, we describe the design of our approach. Given an ActiveX control, we aim to identify and prevent the API-misuse vulnerability in it. We assume we have the knowledge and test cases performing normal functionality of the ActiveX control. This can be obtained from software examples or software documentation. Vulnerabilities such as memory-error exploits are orthogonal and not in our scope.

2.1 ActiveX API-Misuse Vulnerability Detection Phase

In this phase, we identify API-misuse vulnerabilities by building a simple model and analyzing the model as shown in Fig. 2. We first build models to represent the normal functionality of ActiveX methods. This step takes an ActiveX control and test cases invoking the methods in the ActiveX control as the inputs. The output is a model consisting of all function calls in the execution of the program. We analyze the model by searching for a path from ActiveX methods to dangerous APIs. This step takes the model and a dangerous APIs specification as inputs. It outputs a list of vulnerable methods that may result in API misuse.

Model Representation. Our model uses a call graph-based representation. A node in the graph represents a function, which may be called during execution. Every directed edge in the graph (which is labeled) represents the invocation from a caller to a callee function. The edge labels contain information about the order, in which a function is called during execution. More specifically, we classify functions in the model into four types. *Dangerous APIs* are a set of APIs that may lead to more privileges than intended for a web page such as system APIs which expose system resources, e.g., process creation or file operations. There are two types of functions in an ActiveX control. The ActiveX functions that are defined as the scriptable interfaces are the exposed APIs in the ActiveX control, which we call *ActiveX methods*. We call the rest of the functions in the ActiveX control as *ActiveX inner functions*. The functions which are neither ActiveX methods, inner functions nor a dangerous APIs are called *other functions*.

The goal of the model is to gather information on the functionality of an ActiveX control and show the APIs an ActiveX method can reach through paths which represent potential sequences of function calls. There are two types of paths to dangerous APIs in the model. A *direct access path* of an ActiveX method m is when function m called from a webpage has a path to a dangerous API. There may be ActiveX inner and other functions along that path, such that m is in the path and there are no ActiveX methods or ActiveX inner functions in the sub-path from root node to m. An *indirect access path* through ActiveX inner function f is a path from the root node to dangerous APIs such that f is in the path and there are no ActiveX methods or inner functions in the sub-path from root node to f. An example of an indirect access path is as follows, suppose IE calls a callback function f which is an inner function in the ActiveX control, this means that no ActiveX method has been called although an earlier call of some ActiveX method may have returned the address of f. In summary, our model defines API-misuse of an ActiveX method m as either a direct access path from m to a dangerous API or an indirect access path from ActiveX inner function f caused by a ActiveX method m (these may be a set).

Model Building. The goal of this step is to extract the model of an ActiveX control from execution of several test cases. The test cases are meant to be representative of the normal functionality of an ActiveX control. The model is then extracted by instrumenting the execution of the program (IE) running various (standard) test cases.

Direct access paths are straightforward to detect. The challenge is to correlate an ActiveX method to an indirect access path. We modify the model by adding pseudo edges, once an indirect access path is found at an ActiveX inner function f. A conservative approach is to add pseudo edges from every ActiveX method in the corresponding ActiveX control to node f. More accurate dynamic or static analysis could be used to reduce the set of pseudo edges from m but such analysis is necessarily conservative. As static or dynamic analysis can be challenging in the multi-threaded Windows environment which also has the possibility of kernel callbacks.[2] Instead, we chose a simple approach which is to add pseudo edges from the ActiveX methods which were executed prior to the indirect access path.

[2] Non-local control flow transfers where the Windows kernel calls code in the program, somewhat analogous to signals in Unix but are not due to exceptions.

API-misuse Identification. We identify the API misuse in the ActiveX control by searching for access paths in the model. First, we predefine several categories of (system) APIs as dangerous APIs in the model, e.g., file, process creation and library loading APIs. These system APIs allow access to system resources which are normally not exposed to scriptable ActiveX interfaces in IE. ActiveX methods which can have API misuse are found by checking whether an access path exists from the ActiveX methods in the test cases to the dangerous APIs.

2.2 Proxy-Based ActiveX API-Misuse Vulnerability Prevention Phase

To prevent API misuse vulnerability in ActiveX interfaces, we propose a fine-grained proxy-based solution blocking only dangerous ActiveX methods rather than the whole control. We intercept every ActiveX method invocation in the browser and reject any invocation of methods in a blacklist at run time. This blacklist is either generated by our detection phase or defined by users. Rejected methods raise an `E_ACCESSDENIED` exception, i.e. *General Access Denied* exception used in Windows to block access to certain functionality. The advantage of the exception mechanism is that it does not affect the use of other methods in the browser, thus, the user can still interact with a webpage in IE using ActiveX controls as long as dangerous methods are not needed. Additional policies can allow specified trusted webpages to still use methods in the blacklist.

3 Implementation

We have prototyped our approach on Microsoft Windows XP SP2. Our API-misuse detection tool is a PIN tool [1] to collect the function call/return control flow which is then subsequently analyzed to build the model and find API-misuse paths.

In order to track control flow in the program executable and ActiveX binaries which are dynamically loaded, we instrument all binaries during execution. Our PIN tool is an adaptation from [2], which keeps track of how function call and return control flow happens during execution.

To make it more efficient to search for the ActiveX method corresponding to an indirect access path, we use a testing strategy which tests one ActiveX method at the time where possible. This assumes that there is a causal relationship between the single ActiveX method m and any inner function f found in an indirect access path. This can be extended to allow more than one ActiveX method in the test case, in which case, the assumption becomes more relaxed since the causal relation may or may not hold.

In some cases, an ActiveX method can expose objects of another ActiveX to IE, giving a web page the complete access to all methods in the exposed control. This is a more dangerous type of API misuse which is similar to dynamic library loading. To identify this type of API misuse, we apply heuristics to analyze the library loaded by an ActiveX method to identify whether an ActiveX method controls which library to load by specifying an argument – this is a form of data dependency checking.

Table 1. Number of methods with critical access paths in six ActiveX controls

ActiveX Controls	Total methods	file operation	library loading	process creation	access paths
MS ADODB Stream	26	2	0	0	2
MS RDS DataSpace	3	0	1	0	1
MS Office Snapshot Viewer	27	2	3	0	3
Chilkat Crypt	159	2	1	0	2
InstallShield Update Service	14	6	3	3	8
Zenturi ProgramChecker	23	9	4	0	9

4 Evaluation

We evaluated our approach on six real-world ActiveX controls in IE6 SP2: *Microsoft ADODB Stream* (ADODB), *Microsoft RDS DataSpace* (RDS), *Microsoft Office Snapshot Viewer* (Snapshot), *Chilkat Crypt* (Chilkat), *InstallShield Update Service* (InstallShield) and *Zenturi ProgramChecker* (Zenturi). All ActiveX controls have exploits in the Metasploit framework, and have documented functionality except for InstallShield and Zenturi.

4.1 Effectiveness of API-Misuse Vulnerability Detection

Our evaluation uses test cases constructed from user manuals and MSDN library in JavaScript or VBScript for the documented ActiveX controls. For the two ActiveX controls without documentation, we created simple test cases where the parameters to ActiveX methods are initialized to fixed values according to their type. We defined the following three types of dangerous APIs in our evaluation: file operations (`NtCreateFile` and `NtWriteFile`), process creation (`NtCreateProcessEx`), and library loading (`LoadLibraryExW`).

Table 1 shows the results of testing the ActiveX controls. For each control, we list the total number of the exposed methods and number of the exposed methods that have API-misuse access paths found. The access paths are further broken down according to whether they involve file, library or process APIs. Some methods may have multiple access paths, so totaling the number of access paths in individual categories may exceed the total number of methods with access paths. We were successful in detecting API misuses in all six controls which have known API-misuse vulnerabilities and exploits in Metasploit. We identified 25 access paths in total of which seven are indirect access paths which employ callbacks. We now discuss three representative cases in our evaluation.

The Snapshot Viewer ActiveX Control has 27 methods that are available to IE. The generated model has 4963 nodes. We identified three methods have access paths to the system APIs. The `SnapshotPath` and `CompressedPath` methods specify the path to the snapshot file to be displayed in the Snapshot Viewer. We found both have access paths to `NtCreateFile` and `NtWriteFile` with a local or remote URL as the `Path`. With a local URL, `CompressedPath` has a direct access path to `NtCreateFile` and `NtWriteFile`. With a remote URL, `SnapshotPath` has an indirect access path to the

same APIs and we identified the callback functions used in the indirect access path. Other than file operations, `PrintSnapshot` together with the previous methods also has a direct access path to `LoadLibraryExW`.

For the ADODB Stream ActiveX control, we found two methods have direct access paths to `NtCreateFile`: (i) the `SaveToFile` method saves the binary contents of a *Stream* object to a file; and (ii) `LoadFromFile` loads the contents of an existing file into a *Stream* object. We detected the API-misuse vulnerability with the `SaveToFile` method, which has the same direct access path as in the Metasploit sample attack. In Windows, `NtCreateFile` is required to open a file, so `LoadFromFile` is a false positive.

For the RDS DataSpace ActiveX Control, we found that the `CreateObject` method can load any library and create the object registered in the local system through access paths to `LoadLibraryExW`. In the sample exploit, `CreateObject` is used to create objects, from other disabled vulnerable ActiveX controls in IE. It is interesting that this attack bypasses the checking mechanism for preventing certain ActiveX controls from being loaded in IE. The newly created vulnerable object can be further exploited to achieve remote code execution. This exploit has the same access path to `LoadLibraryExW` found in the test case we analyzed. This is also the vulnerability reported in Microsoft advisory `MS06-014` where the killbit checking of IE is bypassed to allow a blocked library to load.

4.2 Performance Evaluation

To evaluate detection performance, we selected 27 test cases from Office Snapshot Viewer ActiveX control. Each method is tested separately in a new IE process. The total time for testing with instrumentation is 1174 seconds (43.5 seconds/test case). The total time for building the model and checking for API-misuse access paths is 264 seconds (9.8 seconds/test case). Although our prototype is not an optimized implementation, it already offers reasonable performance for off-line dynamic analysis.

Our proxy-based prevention mechanism is effective and efficient with negligible overhead. In fact, we target methods that are not in our blacklist with 12 test cases from three ActiveX controls. The overheads range from 0.01% to 1.7%.

5 Related Work

Existing work in ActiveX security mainly focus on memory vulnerabilities in ActiveX controls. Dromann and Plakosh [3] proposed an automated fuzzing system to detect security flaws in ActiveX controls. Its target is memory-related vulnerabilities, instead of API-misuse vulnerabilities. Song et al. [4] proposed an approach to detect malicious exploitation of vulnerable ActiveX controls to prevent drive-by download attacks. The prototype prevention is integrated into IE with ActiveX hooking, using similar techniques as our proxy to block dangerous methods in ActiveX controls.

On a broader problem domain, there are solutions to detect attacks using system-level attack graphs generated by dynamic analysis. For example, Backtracker [5] identifies the files or processes that cause an attack through dependencies between these files

and processes in a system-level dependency graph. As another example, Martignoni et al. [6] perform data-flow analysis to identify high-level actions from system calls.

Our approach is also related to solutions to detect security vulnerabilities using model checking. Schneider [7] proposed security automata for defining security properties and prevent the illegal actions in the system. MOPS [8] detect attacks by checking the reachability of a state that violates the desired security goal in a model. Both Sheyner et al. [9] and Jha et al. [10] construct attack graphs for model checking to detect safety violation in the system.

6 Conclusion

In this paper, we present a system to detect and prevent ActiveX API misuse vulnerabilities in IE. Our system detects potential ActiveX API misuse in an ActiveX control used by IE. Our method can also be easily adapted to other applications using ActiveX. We also provide a prevention mechanism which blocks the use of dangerous ActiveX methods. The results are promising, as we are able to detect all API misuse vulnerabilities in the six real-world ActiveX controls and can block the vulnerable ActiveX methods. The cost of the detection is reasonable and the overhead of the prevention mechanism is negligible.

References

1. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 190–200 (2005)
2. Wu, Y., Yap, R., Ramnath, R.: Comprehending Module Dependencies and Sharing. In: ACM/IEEE Intl. Conf. on Software Engineering, pp. 89–98 (2010)
3. Dormann, W., Plakosh, D.: Vulnerability Detection in ActiveX Controls through Automated Fuzz Testing (2008), http://www.cert.org/archive/pdf/dranzer.pdf
4. Song, C., Zhuge, J., Han, X., Ye, Z.: Preventing Drive-by Download via Inter-Module Communication Monitoring. In: ACM Symp. on Information, Computer and Communications Security, pp. 124–134 (2010)
5. King, S., Chen, P.: Backtracking Intrusions. ACM Trans. on Computer Systems 23(1), 51–76 (2005)
6. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.: A Layered Architecture for Detecting Malicious Behaviors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 78–97. Springer, Heidelberg (2008)
7. Schneider, F.: Enforceable Security Policies. ACM Trans. on Information and System Security 3(1), 30–50 (2000)
8. Chen, H., Wagner, D.: MOPS: an Infrastructure for Examining Security Properties of Software. In: ACM Conf. on Computer and Communications Security, pp. 235–244 (2002)
9. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.: Automated Generation and Analysis of Attack Graphs. In: IEEE Symp. on Security and Privacy, pp. 273–284 (2002)
10. Jha, S., Sheyner, O., Wing, J.: Two Formal Analyses of Attack Graphs. In: IEEE Computer Security Foundations Workshop, pp. 49–63 (2002)