Modeling the trust boundaries created by securable objects

Matt Miller
Leviathan Security Group

Abstract

One of the most critical steps of any security review involves identifying the trust boundaries that an application is exposed to. While methodologies such as threat modeling can be used to help obtain this understanding from an application's design, it can be difficult to accurately map this understanding to an application's implementation. This difficulty suggests that there is a need for techniques that can be used to gain a better understanding of the trust boundaries that exist within an application's implementation.

To help address this problem, this paper describes a technique that can be used to model the trust boundaries that are created by securable objects on Windows. Dynamic instrumentation is used to generate *object trace logs* which describe the contexts in which securable objects are defined, used, and have their security descriptor updated. This information is used to identify the data flows that are permitted by the access rights granted to securable objects. It is then shown how these data flows can be analyzed to gain an understanding of the trust boundaries, threats, and potential elevation paths that exist within a given system.

1 Introduction

One of the most critical aspects of any application security review is the process of modeling an application's trust boundaries. This knowledge allows an auditor to understand how domains of trust are able to influence one another. Without this knowledge, an auditor is generally not be able to easily identify the components of an application that may be exposed to untrusted data. As such, an application's trust boundaries must be understood in order to accurately characterize the threats that exist.

A common methodology that can aide this process is *threat modeling* which has been popularized by Microsoft in recent years as a component of the Security Development Lifecycle (SDL)[10]. Threat modeling provides an auditor with a framework for describing and reasoning about the trust boundaries that exist within an application's design. While threat modeling can help provide an understanding of an application's as-designed security, it is not as adept at providing an understanding of an application's as-implemented security. For instance, an auditor may find it difficult to use threat modeling to express the trust boundaries that are created based on artifacts of an implementation. These deficiencies point to a need for techniques that can be used to improve an auditor's understanding of the trust boundaries that exist within an application's implementation.

A good example of an implementation artifact that can be difficult to capture from an application's design is the way in which an application interacts with *securable objects* such as files, events, and processes. Securable objects, shortened to *objects* henceforth, are used by Windows to provide an abstraction for various resources[12]. Each object is an instance of an *object type* and can be assigned a security descriptor. A security descriptor is used by Windows to describe the access rights *security identifiers* (SIDs) are granted or denied to a given object. These rights can be mapped to the influences each SID may have on other SIDs when using a given object. For example, the ability for all users to write to a registry key whose values are read by an administrative user makes it possible for all users to pass data to, and thus influence, an administrative user. This paper provides an approach that can be used to model and reason about these influences in terms of the data flows permitted by securable object access rights.

The approach described in this paper is composed of two parts. The first part (§2) involves using dynamic instrumentation to generate *object trace logs* which provide the raw data needed to analyze the data flows permitted by securable object access rights. The second part (§3) involves interpreting the object trace log data to analyze data flows, trust boundaries, threats, and potential elevation paths.

1.1 Contributions

The primary motivation for this paper is to provide tools and techniques needed to allow a security auditor to model and reason about the trust boundaries created by securable objects. While there are many other types of trust boundaries that can exist within an application, such as those related to network connectivity and system calls, this paper focuses strictly on securable objects. In this vein, the specific contributions in this paper include:

- A technique that can be used to dynamically instrument securable object definitions and uses on Windows.
- A model that can be used to represent the data flows permitted by securable object access rights, the trust boundaries that are traversed, and the threats that exist as a result.

1.2 Related work

There has been significant work done on developing software verification techniques that focus on determining that a program satisfies a given specification[4, 2, 15, 8]. These techniques have also been applied to help support software security analysis[6, 3, 10, 1]. The work presented in this paper may aide software security analysis by automatically deriving an understanding of the data flows permitted by securable object access rights as obtained from a program's implementation. For example, this understanding could be used to help provide raw data for determining threat model conformance with Reflexion models[1].

Previous work has also shown that specific instances of privilege escalations can be detected by using a logical model of the Windows access control system to analyze the access rights assigned to persistent file, registry key, and service objects[14, 7]. This paper extends this work by using dynamic instrumentation to collect access control information for all securable object types.

2 Data collection

The access rights associated with a given object define the extent to which each SID can influence one another. As such, the first step to understanding these influences involves collecting access right information for all objects. This paper uses a combination of two approaches to obtain this information for both persistent and dynamic objects. The data collected by both approaches is ultimately written to one or more *object trace logs* which provide the raw input used in §3.

2.1 Persistent objects

Persistent objects are non-volatile objects that may exist before a system has booted. The most prevalent persistent objects are files, registry keys, and services. The access rights associated with each of these objects can be obtained using functionality provided by the Windows API[12]. Specifically, `GetNamedSecurityInfo` can be used for files, `GetSecurityInfo` can be used for registry keys, and `QueryServiceObjectSecurity` can be used for services. It is suspected that this approach likely mirrors that which was used in previous work[7].

2.2 Dynamic objects

Dynamic objects include both volatile and non-volatile objects that are defined while a system is executing. Examples of dynamic objects include sections, events, and processes. The access rights associated with dynamic objects can be obtained by dynamically instrumenting the object manager in the Windows kernel. Dynamic instrumentation makes it possible to collect information about the contexts in which objects are defined, used, and have their security descriptor updated. This information can be used to better distinguish between access rights that *can* be granted to a SID, such as by a security descriptor, and access rights that *are* granted to a SID, such as when a SID defines or uses an object.

2.2.1 Object definitions

Every object that is defined on Windows must first be allocated and initialized by `ObCreateObject`. By instrumenting `ObCreateObject`, it is possible to log information about the context that defines a given object. This contextual information includes the calling process context, active security tokens, initial security descriptor (if present), and call stack. It is implicitly assumed that the SID responsible for defining an object is granted full rights to the object.

2.2.2 Object uses

When an application uses an object it must typically create a handle to the object by issuing a

call to a routine that is specific to a given object type, such as `NtOpenProcess`. These routines ultimately make use of functions provided by the object manager, such as `ObLookupObjectByName` and `ObOpenObjectByPointer`, to actually acquire a handle to the object. While there are many places that could be instrumented, there are two options that have superior qualities: hooking the `OpenProcedure` of each object type or using object manager callbacks. This paper will only discuss the use of object manager callbacks.

Object manager callbacks are a new feature in Windows Vista SP1 and Windows Server 2008[11]. This feature provides a new API, `ObRegisterCallbacks`, which allows device drivers to register a callback that is notified when handles to objects of a given type are created or duplicated. While this API would appear to be the perfect choice, the default implementation only allows callbacks to be registered for process and thread object types (`PsProcessType` and `PsThreadType`). Fortunately, this limitation can be overcome by dynamically altering a flag associated with each object type which enables the use of `ObRegisterCallbacks`. Once registered, each callback is then able to log information about the context that uses a given object such as the calling process context, active security tokens, assigned security descriptor, granted access rights, call stack, and object name information.

2.2.3 Object security descriptor updates

The `SecurityProcedure` function pointer associated with each object type must be instrumented in order to detect alterations to an object's security descriptor. The `SecurityProcedure` of an object type is called whenever an object's security descriptor is modified during the course of an object's lifetime, such as through a call to `SetKernelObjectSecurity`. This allows each instrumented security procedure to log information about the security descriptor that is assigned to individual objects during the course of execution.

2.2.4 Memory mapped images

In addition to instrumenting object definitions, uses, and security descriptor updates, it is also helpful to instrument the memory mapping of loadable modules. This can be accomplished by using `PsSetLoadImageNotifyRoutine` to register a callback that is notified whenever a module is mapped into the address space of a given process. This gives the callback an opportunity to log information about the base address and image size of each mapping as an attribute of a process object. Memory mapping information is needed in order to determine what image file the return address of a given call stack frame is contained within when interpreting call stack information.

3 Trust boundary analysis

Object trace log data can be used to better understand how SIDs are able to use objects to influence one another. These influences can be modeled by describing the flow of data between SIDs as permitted by the access rights each SID is granted to a given object. To illustrate this, §3.1 shows how a *data flow graph* (DFG) can be used to describe the permitted data flow behavior of a system. §3.2 then describes how a DFG can be generated by interpreting object trace log data. Finally, §3.3 shows how trust boundaries and threats can be derived from a given DFG.

3.1 Definitions

A data flow graph $G = (D, U, E)$ relates a data definition context $d \in D$ with a data use context $u \in U$ such that $du \in E$. Each vertex is defined as a tuple $d, u = \langle a, m, v \rangle$ where a is an actor that belongs to a domain of trust, m is a medium through which data is flowing, and v is a verb that describes how data is transferred. Each verb is defined as $v = \langle n, C, T \rangle$ where n is the name of the verb, C is a set of matching criteria, and T is a set of threats posed to an actor that makes use of the verb. A *data flow* $du \in E$ exists whenever d and u are using complementary verbs $v_d v_u \in V$ to operate on related mediums $m_d m_u \in M$.

In the context of this paper, each vertex in a DFG takes on a more precise definition. Specifically, a represents a SID or a group of SIDs, m represents an object instance, and v represents a verb that is specific to the object type of m. A verb's matching criteria C describes the access rights that must be granted for a SID to operate on an object. Using these definitions, a data flow exists whenever d and u are using complementary verbs, such as those found in figure 1, to operate on the same medium $m_d = m_u$. For example, a definition $d = \langle$`S-1-1-0`$, \mathit{LsaPort},$ `Write request`$\rangle$ and a use $u = \langle$`S-1-5-18`$, \mathit{LsaPort},$ `Read request`$\rangle$ illustrates a data flow where `S-1-1-0` writes a request to *LsaPort* which is then read by `S-1-5-18`.

Object Type	v_d			v_u		
	Name	**Criteria**	**Threats**	**Name**	**Criteria**	**Threats**
ALPC Port	Write request	`CONNECT`	I	Read request	`IMPLICIT_DEF`	STRIDE
	Write reply	`IMPLICIT_DEF`	I	Read reply	`CONNECT`	STRIDE
File	Write data	`WRITE_DATA`	I	Read data	`READ_DATA`	STRIDE
	Write data	`WRITE_DATA`	I	Execute process	`EXECUTE`	STRIDE
Key	Set value	`SET_VALUE`	I	Query value	`QUERY_VALUE`	STRIDE
Process	Write memory	`VM_WRITE`		Execute code	`IMPLICIT_USE`	STRIDE
	Terminate process	`TERMINATE`		Kill process	`IMPLICIT_USE`	D
	Create thread	`CREATE_THREAD`		Execute code	`IMPLICIT_USE`	STRIDE
Section	Write memory	`MAP_WRITE`	I	Read memory	`MAP_READ`	STRIDE
	Write memory	`MAP_WRITE`	I	Execute memory	`MAP_EXECUTE`	STRIDE
Service	Change config	`CHANGE_CONFIG`		Start service	`IMPLICIT_USE`	STRIDE
Thread	Set context	`SET_CONTEXT`		Execute thread	`IMPLICIT_USE`	STRIDE

Figure 1: **Verb relationships $v_d v_u \in V$ for a subset of the object types that exist on Windows. These relationships describe how data can flow through an object where v_d defines data that is used by v_u. Matching criteria are expressed in terms of Windows access rights required to make use of the verb. Threats are expressed categorically using STRIDE[10].**

3.2 DFG generation

A DFG can be generated by interpreting object trace log records that contain information about the access rights individual SIDs are granted to each object. This information exists in log records that describe when an object is defined, used, or has its security descriptor updated.

When an object is defined, a vertex is created for each verb associated with the object's object type, with the exception of verbs having the criteria `IMPLICIT_USE`. This is meant to capture the fact that the definer of an object is implicitly granted full access to the object and is thus capable of using all verbs. Each vertex is created in terms of the context that defined the object where a is either the `Owner` of the object's security descriptor or the active security context's client token or primary token owner SID, m is the object being defined, and v is the verb whose criteria was satisfied. For example, the dynamic definition of an ALPC port object would lead to the creation of a vertex $u = \langle \texttt{SID}, object, \texttt{Read request} \rangle$.

When an object is used, a vertex is defined for any v that matches the access rights granted to the SID that uses the object. The owner SID and the primary group SID of either the client or primary thread token represent the actors that are involved. For example, acquiring a handle to a registry key with granted rights of `KEY_SET_VALUE` would lead to the creation of a vertex $d = \langle \texttt{SID}, object, \texttt{Set value} \rangle$.

When an object's security descriptor is assigned or updated, zero or more vertices may be created as a result. Log records that provide information about an object's security descriptor can be interpreted by enumerating the *access control entries* (ACEs) contained within the *discretionary access control list* (DACL) of the object's security descriptor. Each ACE contains information about the rights granted to a SID for a given object. A security descriptor with a *null* DACL can be interpreted as granting full access to all SIDs. If the rights granted to a SID meet the criteria of a given v then a vertex can be defined where the actor is the SID that is derived from the corresponding ACE. For example, an ACE that grants the SID `S-1-5-18` the `KEY_QUERY_VALUE` access right would lead to the creation of a vertex $u = \langle \texttt{S-1-5-18}, object, \texttt{Query value} \rangle$.

The vertices that are created as a result of this process can be combined together to form data flows as described in §3.1. A data flow may also be created in circumstances where the verb of a given vertex is related to a verb having the criteria `IMPLICIT_USE`. This criteria captures behavior that implicitly follows from a definition. For example, the act of writing data into a process address space can implicitly lead to the execution of the injected data as code.

3.3 DFG analysis

Once generated, a DFG can be analyzed to derive a number of properties including the set of trust boundaries that exist, the potential threats posed to each domain of trust, and the risks posed to specific regions of code.

3.3.1 Trust boundaries

For the purpose of this paper, a *trust boundary* is defined as a medium, m, that allows data to flow between domains of trust. The set of trust boundaries $S = \{m_1, m_2, \ldots\}$ that exist within a DFG can be derived from the subset of data flows where the definition and use actors are not equal such that $m_d, m_u \in S$ given $\{du \mid du \in E, a_d \neq a_u\}$. In other words, a data flow involving different domains of trust must implicitly cross a trust boundary. The subset of data flows that cross a trust boundary compose a *trust boundary data flow graph* (TBDFG). Figure 2 provides a summary of a TBDFG where each edge conveys the number of data flows, and thus potential elevation paths, involving a_d and a_u.

3.3.2 Threats

The flow of data between domains of trust can lead to threats such as *elevation of privilege* and *denial of service* as categorized by `STRIDE`[10]. Determining which data flows pose a threat is entirely dependent on the perspective of a domain of trust. In the following descriptions, the relation operator $\prec$ can be interpreted as *less privileged than*.

From a defensive perspective, a *defense horizon* can provide an understanding of the threats posed *to* a given domain of trust, a. A defense horizon is composed of the subset of data flows which may result in other domains of trust threatening a with a set of threats T[1]. This is captured by $\delta(a, T) = \{du_1, du_2, \ldots\}$ given $T \cap T_{v_u} \neq \emptyset, a = a_u, a_d \prec a_u$.

Conversely, the *attack horizon* for a domain of trust can provide an understanding of the threats posed *by* a given domain of trust. An attack horizon is composed of the subset of data flows which may result in a threatening other domains of trust with a set of threats T. This is captured by $\alpha(a, T) = \{du_1, du_2, \ldots\}$ given $T \cap T_{v_u} \neq \emptyset, a = a_d, a_d \prec a_u$.

3.3.3 Actualized and potential data flows

Data flows can be further classified in terms of whether or not they are *actualized*. An actualized data flow exists whenever the u vertex was created as a result of the access rights granted when an object was dynamically defined or used. On the other hand, a *potential* data flow exists whenever the u vertex was created as a result of the access rights granted by an object's security descriptor.

Actualized data flows are interesting from an analysis perspective because they represent threats that can be immediately acted upon. Potential data flows are more difficult to interpret from an analysis perspective as they may never become actualized. For example, the ability for all users to write to a file that can be executed by an administrator produces a potential data flow with a threat that could allow all users to elevate privileges to administrator. However, this is predicated on the administrator actually executing the file which may never occur in practice.

3.3.4 Assigning risk attributes to code

It is not always easy to determine what code is responsible for exposing a trust boundary when assessing the security of a program. This determination can be made easier by taking into account the call stacks that are logged to an object trace log when an object is defined or used. This data makes it possible to determine how different areas of code contribute to a program's overall risk. This understanding may benefit traditional program analysis by helping to scope analysis to areas of a program with higher risk attributes based on their exposure to a trust boundary. This data could also be used to support security metrics that relate to code exposure[5, 9].

3.4 Applications

To better illustrate how this model can be useful, it is helpful to consider some of the ways in which it can be applied.

3.4.1 Finding privileged ALPC ports

ALPC ports represent a good target for elevation of privilege attacks due to their client-server nature. Figure 3 provides a subset of the ALPC port data flows that compose the defense horizon for `SYSTEM` on a default installation of Windows Vista SP1. When analyzing these data flows it is possible to determine what code was responsible for exposing a given trust boundary by inspecting the call stack that was captured at the time that a server-side ALPC port object was defined. This allows an auditor to quickly identify code that may be at risk. For example, the following call stack lead to the creation of a trust boundary through `\RPC Control\plugplay` from figure 3.

[1]The term *attack surface* also describes this set but is considered less precise.

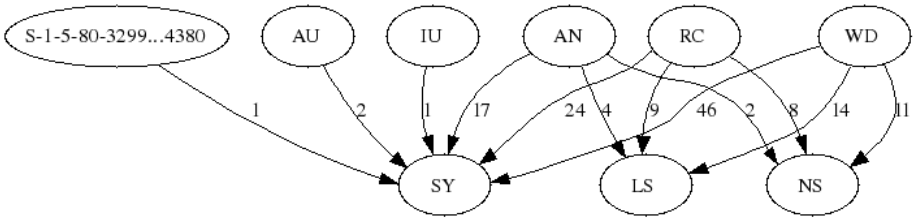

Figure 2: **A summary of the data flows that exist within the TBDFG generated for ALPC Port objects on Windows Vista SP1. Each edge provides a count of the number of data flows where a_d can define data that may be used by a_u. Each vertex represents an actor using SID strings[13]. Informally, this graph illustrates the number of potential elevation paths from a_d to a_u as enabled by ALPC Port objects.**

```
ntoskrnl!AlpcpCreateConnectionPort+0xd0
ntoskrnl!NtAlpcCreatePort+0x29
ntoskrnl!KiSystemServiceCopyEnd+0x13
ntdll!ZwAlpcCreatePort+0xa
rpcrt4!LRPC_ADDRESS::ActuallySetupAddress+0xf8
rpcrt4!LRPC_ADDRESS::ServerSetupAddress+0x90
rpcrt4!RPC_SERVER::UseRpcProtocolSequence+0x1b4
rpcrt4!I_RpcServerUseProtseqEp2W+0x83
rpcrt4!RpcServerUseProtseqEpW+0x35
umpnpmgr!ServiceMain+0x189
svchost!ServiceStarter+0x1ea
advapi32!ScSvcctrlThreadA+0x25
kernel32!BaseThreadInitThunk+0xd
ntdll!LdrpInitializeThread+0x9
```

ID	a_d	m_d
1	WD	\Sessions\1\Windows\ApiPort
2	WD	\RPC Control\trkwks
3	WD	\AELPort
4	WD	\UxSmsApiPort
5	WD	\RPC Control\samss lpc
6	WD	\RPC Control\spoolss
7	WD	\RPC Control\plugplay
8	AU	\WindowsErrorReportingServicePort
9	WD	\LsaAuthenticationPort
10	AU	\BaseNamedObjects\msctf.serverWinlogon1

Figure 3: **The d vertices for a subset of data flows $du \in \delta(\texttt{SYSTEM}, \{E\})$ where $u = \langle \texttt{SYSTEM}, m, \texttt{Read request} \rangle$ and $v_d = \texttt{Write request}$ as exposed by ALPC ports on Windows Vista SP1.**

3.4.2 Using services to elevate privileges

A privilege elevation can occur whenever a low-privileged SID is allowed to change the configuration of a service. As a result, a lesser privileged SID can execute arbitrary code with the privileges of SYSTEM since it is possible to alter the image file of the service and the credentials that the service executes with. This allows a given SID to threaten to elevate privileges to SYSTEM. In other words, a data flow exists such that $d = \langle \texttt{SID}, service, \texttt{Change config} \rangle$ and $u = \langle \texttt{SYSTEM}, service, \texttt{Start service} \rangle$ whenever $\texttt{SID} \prec \texttt{SYSTEM}$. The default installation of Windows Vista SP1 has no data flows that enable this specific elevation path. Previous work has also shown how this elevation path can be detected[7].

4 Conclusion

An application's trust boundaries and data flows must be understood in order to identify relevant threats. Threat modeling is a valuable tool that can be used to help provide this understanding of an application's design. Still, it can be difficult to map this design understanding to an application's actual implementation. This can lead to divergences in one's understanding of the threats that actually exist. It can also impact an auditor's ability to know which components may encounter untrusted data. These deficiencies point to the need for techniques that can help to derive trust boundary information from an application's implementation.

This paper has shown how to model the trust boundaries that are created by securable objects on Windows. Dynamic instrumentation was used to create *object trace logs* which contain information about the contexts in which securable objects are defined, used, and updated. The object trace log data was then used to model and reason about the data flows, trust boundaries, and threats permitted by securable object access rights. Future work will attempt to extend this model to other types of trust boundaries in an effort to gain a more complete understanding of the trust boundaries that exist within a given system.

References

[1] M. Abi-Antoun, D. Wang, and P. Torr. Checking threat modeling data flow diagrams for implementation conformance and security. http://reports-archive.adm.cs.cmu.edu/anon/isri2006/CMU-ISRI-06-124.ps, 2006.

[2] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[3] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. http://www.csl.sri.com/users/ddean/papers/ndss04.pdf, 2004.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[5] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, 1997.

[6] David P. Gilliam, John C. Kelly, John D. Powell, and Matt Bishop. Development of a software security assessment instrument to reduce software security risk. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 144–149, Washington, DC, USA, 2001. IEEE Computer Society.

[7] S. Govindavajhala and A. Appel. Windows access control demystified. http://www.cs.princeton.edu/~sudhakar/papers/winval.pdf, 2006.

[8] Gerard J. Holzmann. Trends in software verification. http://spinroot.com/gerard/pdf/fme03.pdf, 2003.

[9] A. Jaquith. Security metrics: Replacing fear, uncertainty, and doubt. Addison-Wesley, April 2007.

[10] Microsoft. The trustworthy computing security development lifecycle. http://msdn.microsoft.com/en-us/library/ms995349.aspx, 2005.

[11] Microsoft. Kernel data and filtering support for vista sp1. http://download.microsoft.com/download/4/4/b/44bb7147-f058-4002-9ab2-ed22870e3fe9/Kernal%20Data%20and%20Filtering%20Support%20for%20Windows%20Server%202008.doc, 2007.

[12] Microsoft. Securable objects (windows). http://msdn.microsoft.com/en-us/library/aa379557(VS.85).aspx, 2008.

[13] Microsoft. Sid strings. http://msdn.microsoft.com/en-us/library/aa379602(VS.85).aspx, 2008.

[14] Prasad Naldurg, Stefan Schwoon, Sriram Rajamani, and John Lambert. Netra:: seeing through access control. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 55–66, New York, NY, USA, 2006. ACM.

[15] Martin Ouimet. Formal software verification: Model checking and theorem proving. http://esl.mit.edu/publications/ESL-TIK-00214.pdf.