Security Relevancy Analysis on the Registry of Windows NT 4.0

Wenliang Du
CERIAS*
Computer Sciences Department
Purdue University
West Lafayette, IN 47907
duw@cs.purdue.edu

Praerit Garg
Microsoft Corporation
Redmond, WA 98052
praeritg@microsoft.com

Aditya P. Mathur
Computer Sciences Department
Purdue University,
West Lafayette, IN 47907
apm@cs.purdue.edu

Abstract

Many security breaches are caused by inappropriate inputs crafted by people with malicious intents. To enhance the system security, we need either to ensure that inappropriate inputs are filtered out by the program, or to ensure that only trusted people can access those inputs. In the second approach, we sure do not want to put such constraint on every input, instead, we only want to restrict the access to the security relevant inputs. The goal of this paper is to investigate how to identify which inputs are relevant to system security. We formulate the problem as an security relevancy problem, and deploy static analysis technique to identify security relevant inputs. Our approach is based on dependency analysis technique; it identifies if the behavior of any security critical action depends on certain input. If such a dependency relationship exists, we say that the input is security relevant, otherwise, we say the input is security non-relevant. This technique is applied to a security analysis project initiated by Microsoft Windows NT security group. The project is intended to identify security relevant registry keys in the Windows NT operating system. The results from this approach is proved useful to enhancing Windows NT security. Our experiences and results from this project are presented in the paper.

1 Introduction

To build a secure system, it is important to understand system behaviors, especially those behaviors that respond to inputs; to understand those behaviors, knowing whether an input is *security relevant* is important. The *security relevancy* of an input is defined based on the definition of a security critical action. A security critical action is an action, which, if conducted in a uncontrolled manner, can compromise system security. For example, in UNIX, `system()` is a security critical action since it invokes a command, which could be any command if the argument passed onto `system()` is not appropriately controlled. Generally speaking, an input is *security relevant* if the data from this input will affect the behavior of at least one security critical action. A formal definition is given in section 2.

There are many different kinds of inputs to a program. The most obvious ones are the input from users. Less obvious ones are inputs from files, from network, from environment variables, from other processes, or from the Windows NT Registry. Some of these are critical to system security, some are not. By saying "critical to system security", we mean that if the input data is validated incorrectly or the validation is missing, the system security could be compromised by the manipulation of the input in certain way.

Let us take Windows NT Registry as an example. Windows NT Registry is essentially an organized storage for operating system and application data. This data is globally shared by different applications and different components of the operating system. Please see section 3.1 for the definition of *Registry* and *registry key* terminology. When a program gets data from the Registry, the data now becomes an input, and some of this input are benign while some are not. For example, in one scenario the program gets an input from a registry key and treats this input as a file name, then displays to the user this input in a message window. Even if somebody can arbitrarily manipulate the data, no harm will be done to system security itself (though the message can be changed in such a way that the user is tricked to do something harmful). In another scenario, the data retrieved from the registry key is still treated as a file name, but the program proceeds to execute the file represented by this name, This input now becomes a dangerous input, which means leaving the source of the input (the registry

*Center for Education and Research in Information Assurance and Security (CERIAS)

key in this case) unprotected or using the input without an appropriate validation might now lead to a security breach.

Consequently, knowing which inputs are critical to system security is essential to enhancing system security. In the previous Windows NT Registry example, knowing that a registry key is critical to system security will enable us to put a protection on that key to prevent unauthorized modification. However, there is no easy way to know that. Furthermore, one protection configuration might become invalid in the new version of the operating system because changes to the code could make a security non-critical registry key become security critical, and vice versa. It is not always obvious to identify which part of the configuration is not valid any more since people who made the decision that certain keys are security critical might have left the company without leaving the corresponding documentation on why that protection decision was made. From discussions with NT developers, we have learned that they are constantly looking for the reasons why they have put some registry keys into protection mode. Their customers, after all, want to know whether they themselves should really put certain keys under protection or not. Sometimes, they may decide to put less restriction on certain keys, but they want to know how much risk that would bring to the system. Moreover, every time developers made a major revision on the operation system program, people want to know whether those reasons still stand.

Knowing whether an input is critical to system security is also important to security testing. It can help testers allocate their resources wisely. The key difference between secure software and other high quality software is that secure systems have to be able to withstand active attacks by potential penetrators. When developing a secure system the developers must assure that intentional abnormal actions can not compromise the system. In another words, secure systems must be able to avoid problems caused by malicious users with unlimited resources [9]. Knowing that an input is not critical to system security, testers do not need to spend time in designing attacks against that specific input; instead, they can focus on those inputs that are critical to system security. Furthermore, knowing that an security critical action depends on the value of an input provides testers with more information for security testing cases. If they know, for instance, that the value of an input is treated as a file name and is subject to execution, their test cases would thus involve using files with different properties, permissions, owners etc. We have developed an environment perturbation technique based on this knowledge in [4].

Knowing whether an input is critical to system security is not trivial. It is not sufficient to just look at the content of an input. In the example used before, the input data in both case are exactly the same (file names), but they are used for different purposes, thus implicating different consequences. How can we identify their purposes?

This problem can be formulated as a dependency problem [7]. An example can help illustrate this point. As is known, in UNIX, `system` action is a security critical action, the consequence of which depends on the value of the actual argument passed to this action. If the action takes the form of `system("rm /etc/passwd")`, it will erase `/etc/passwd` file, which will cause a severe security problem. But, if the action takes the form of `system ("ls")`, it will not do as much harm as the former action. From this perspective, the value of the actual argument passed into the `system` action actually decides the security consequence. The value in the actual argument can be affected by various sources. If an input is one of these sources, we say that this `system` action depends on the input and thus the input is considered security relevant. Dependency relationship exactly models the correlations among various variables. If variable a affects variable b's value, we say b depends on a. Therefore, to find out if an input is security relevant to system security is equivalent to finding out the dependency relationship among the program's variables, especially dependency relationships between arguments sent to a security critical action and variables that represent inputs.

Dependency analysis technique has already been used in detecting a variety of anomalies in program, in testing, and in program slicing [7]. The work presented here appears to be the first attempt to detect security relevancy of inputs using dependency analysis. Also presented is our experience with the application of this technique to Windows NT 4.0 source code.

In addition to static analysis, another possible way of identifying this kind of dependency relationship is to derive it from design specification. By analyzing specification, one can understand how the program will use the input data. This, to some extent, can generate more precise information about the dependency. However, this is not always feasible. In reality, many inputs are hidden from the design specification because it belongs to implementation details. For example, inputs from files or from the Registry are frequently hidden from design specification, and thus learning the security relevancy of these hidden inputs is impossible from specification analysis. Another drawback via this approach is the difficulty of automation unless the specification is written in a strictly formal language.

The remainder of this paper is organized as follows. Section 2 describes the dependency and security-relevancy analysis. Section 3 presents the application of the security-relevancy analysis on windows NT 4.0 source code. Section 4 briefly reviews related works in this research area.

Finally, section 5 draws conclusions and points out future work.

2 Analysis

This section describes dependency analysis technique and based upon which, we will discuss security relevancy analysis.

2.1 Dependency analysis

Dependency analysis has been discussed in several works [18], [8], [7], [13], [16]. However, most of those works focus on finding data and control dependency relationships among statements. We, however, discuss a similar technique to identify dependency relationships among variables.

A program P has a dependence relation D among its variables

$$D(P) : Var \leftrightarrow Var$$

where a pair $(x, y) \in D(P)$ means that the value of the variable x, after execution of P, depends on the value of y before execution of P. Each of such pair represents a dependency relationship in the program P.

To specify the dependency relationship formally, we borrow the notation from [7]: Representing the behavior of program P as a function p over some set of program variables like a, b, c, etc.

$$p : (a, b, c, ...) \rightarrow (a, b, c, ...)$$

we say that variable x depends on variable y when there are two prestates s and s' that are distinguishable only in their y components and lead, under P, to corresponding post-states having different x components:

$$(x, y) \in D(P) \text{ iff } \exists s, s'.\ \forall v \neq y.$$
$$s|v = s'|v \cap p(s)|x \neq p(s')|x.$$

(Here $s|v$ means the value of variable v in state s.) In other words, x depends on y if the computation of x uses y.

A direct dependency relationship is a dependency relationship derived from a primitive statement, which could be assignment or procedure call. A Data Dependency Graph (DDG) could be built based on the direct dependency relationships among variables.

A DDG is actually a directed graph, the node of which represents a variable, and the edge of which represents a direct dependency relationship. If there exists a direct dependency between variables A and B, say B directly depends on A, then in the DDG the relationship is shown as a directed edge from A's node to B's node. Since the dependency relationship is transitive, with DDG the dependency relationship between two variables can be rephrased as the following: a variable x depends on another variable y if and only if there exists a path from y's node to x's node in the Data Dependency Graph. Therefore, for the purpose of capturing dependency relationships among variables, all one needs to do is to build a DDG. We will use $DD(P)$ to represent direct dependency relationships derived from program P.

During the analysis, we will assume that each variable, whether a local variable, global variable or formal parameter, has a different identifier. This can easily be achieved by renaming.

Simple dependence analysis

If two or more variables denote the same memory address, we say that the variables are *aliases* of one another. The presence of pointers makes data-flow analysis more complex because they cause uncertainty regarding what is defined and used [18]. In this part of the analysis, we temporarily suppose that no alias exists in the program; thus, each variable represent a distinguished memory location.

The primitive statement that generates direct dependency relationships is an assignment statement:

$$DD(x = y) = (x, y)$$

A composite statement generates direct dependency relationships in the following way:

$$DD(\text{if } W \text{ then } S \text{ else } T) = DD(W) \cup DD(S) \cup DD(T)$$
$$DD(\text{while } W \text{ do } S) = DD(W) \cup DD(S)$$
$$DD(S; T) = DD(S) \cup DD(T)$$

Now let us analyze dependency relationships among variables across different procedures. As we know, this kind of relationship is caused by inter-procedure call. So, let us use a general form of procedure invocation S: $w = f(x_1, x_2, ...x_m)$. To simplify the discussion, suppose the identifier for the return value of f is r, and the formal arguments of f is $v_1, v_2, ...v_m$.

Since we have supposed that there is no alias type, the data of actual arguments are passed onto formal arguments via pass-by-value, i.e. during the invocation, it actually has a set of assignment statements: $v_i = x_i$, where $i = 1...m$. Therefore, the resultant dependency relationship is:

$$DD(S) = \{(v_i, x_i), \text{ where } i = 1...m\} \cup \{(w, r)\}$$

With alias

If two variables denote the same memory address, namely, they are aliases of one another, the analysis becomes more complicated because the presence of pointers causes uncertainty regarding what is defined and used. An assignment

of `*x = *y` could cause the dependency of u and v if x and y are the aliases of u and v respectively.

The safest assumption is that a pointer p can point to any variable in the program. Thus, a single assignment like `*p = *q` causes a dependency relationship between any two variables. Although a knowledge of variable scope can cut down the number of dependency pairs, the assumption is still too strong for dependency analysis to derive an accurate relationship.

Several methods of alias analysis and point-to analysis have been proposed [12, 6, 19, 2, 20]. By using these methods, one can compute a *points-to* set for each variable. The *points-to* analysis is beyond the scope of this paper, and we assume that a *points-to* set for each variables could be obtained via this analysis. The main concern of this paper is how to use the *points-to* sets to build a Data Dependency Graph, and based on which, how to conduct security-relevancy analysis. In the following analysis, we use $\phi(a)$ to represents the *points-to* set of variable a.

With a *points-to* set for each variables available, one can compute dependency relationships from the following assignments:

$$DD(*p = *q) = \{(x, y) | x \in \phi(p), y \in \phi(q)\}$$
$$DD(*p = v) = \{(x, v) | x \in \phi(p)\}$$
$$DD(u = *q) = \{(u, y) | y \in \phi(q)\}$$

2.2 Incomplete program

An assumption underneath the above analysis is that the source code for a program is complete. However, in practice this assumption is not always true. Library routines, for instance, usually come with no source codes. To solve this problem, a dependency digest for each of those library subroutines is manually computed. A dependency digest of a subroutine represents the dependency relationship among its formal parameters and return value.

For example, `char * strcpy(char *s1, char *s2)` subroutine will copy the contents pointed by $s2$ to the location pointed by $s1$, and return the value of $s1$. Thus the dependency digest is:

$\{(*s1, *s2), (*r, *s1), (r, s1)\}$, where r is the return value.

Therefore, for the statement `S: x = strcpy(a, b)`, we have

$$\{DD(S) = \{(x, a)\} \cup \{(\phi(x), \phi(a))\} \cup \{(\phi(a), \phi(b))\}$$

2.3 Security Relevancy Analysis

Security critical action

Some of actions conducted by a program could be benign while some might be *security critical*, which means that if the target of the action is not verified correctly, the action could lead to breach, such as impairing system integrity, confidentiality, accountability, or availability. Examples of such actions are system calls like `write()`, `unlink()`. Take `write()` as an example: if the target of the write action is not appropriately validated, this operation could be applied to an unwanted target, thus overwriting the target.

In operating systems such as Windows NT, UNIX, a security critical action usually is represented by a system call or by a procedure from library that invokes system calls. A security critical attribute is associated with each of of this kind of procedure indicating whether its invocation has any potential consequence on system security. We define a variable's security relevancy based on these security critical actions.

Definition 2.1 *(Security Relevancy of Variable)* A variable x is security relevant in program P (denoted as $x \in SR(P)$), if one of the following situations is true:

1. x is passed as a parameter passed onto function f, where f is security critical.
2. $(v, x) \in D(P)$ and v is security relevant.

After obtaining the direct dependency relationships among all variables of the program, one can build a Data Dependency Graph (DDG). A DDG is actually a directed graph, the node of which represents a variable, and the edge of which represents a direct dependency relationship. If there exists a direct dependency between variables A and B (say B depends on A), then in the DDG the relationship is shown as a directed edge from A's node to B's node.

We will distinguish those variables which represent inputs from other variables by marking each of their nodes with an I. We will also distinguish the variables which are fed directly to security critical actions from other variables by marking each of their nodes with an S. The rest of variables are marked with an O. Now the problem of determining whether an input is security relevant is transformed into the following problem statement:

Definition 2.2 *(Security Relevancy Problem)* Given the directed graph $G = (V, E)$, where $V = I \cup S \cup O$, and I, S, O are three sets of nodes with different properties, finding all security relevant inputs is equivalent to finding all nodes $i \in I$, such that $\exists s \in S$, and there exists at least a path from i to s,

Proof: Since set S contains all security relevant nodes, and set I contains all input nodes, if there exists a path from an I node to an S node, from the dependency definition, we know that the S node depends on the I node. From the definition of security relevancy of a variable, the I node is

a security relevant variable. The input it represents is thus a security relevant input.

An intuitive solution to this graph problem is to first reverse the direction of each edge, then to find the complete reachable set for each S node, then check whether the set contains any I node. If so, one can decide that the I node is security relevant. A straightforward implementation would have the running time of $O(|S| \times n)$, where n is the number of security relevant variables. In the worse case, where $|S|$ is in the order of n, the algorithm would take $O(n^2)$ time.

An improved algorithm would (1) reverse the direction of each edge like the above solution; (2) choose a node s from S set, find the reachable set for node s; (3) delete all nodes that are in this reachable set from the graph, as well as all the edges connected to these nodes; (4) choose another unchosen node from S set, and repeat step (2) until there are no more nodes to choose. Finally, if any node from I appears in the union of all reachable sets, we say that the node is security relevant. Since the improved algorithm only traverses each security relevant node once, the total running time would be $O(n)$, where n is the number of security relevant variables.

To further increase the performance of the algorithm, one could compress the Data Dependency Graph to some extent. For example, once a set of dependency relationships for each procedure is obtained, all relationships among local variables could be removed if they are not related to any input. Thereby, only the dependency relationships among parameters, global variables, and input-related local variables are kept. Of course, one can not simply get rid of those local variables, since, for example, some formal parameter might depend on a local variable, which itself depends on another formal parameter, This circumstance makes the first formal parameter depend on the second one. The indirect dependency relationships among formal parameters and global variables should be preserved while the dependency relationship set is reduced.

3 Registry Security Analysis Project

3.1 Background of the Project

The Registry in Windows NT 4.0 is laid out in a hierarchical structure of *keys* and *name-value pairs*. This structure is used as a central configuration database for the user, application, operating system, and computer information. A **key** is a node of the hierarchical Registry structure. It consists of sub-keys and name-value pairs. A **sub-key** is the child of a parent key. A **name-value pair** is the holder of the data within a registry key. Each key may have any number of sub-keys and/or name-value pairs [3]. We will use registry key/value in this paper to refer to both key and name-value pairs.

Definition 3.1 *Security relevant registry key:* a registry name-value is security relevant if a change in its value in some way could lead to violation of system security, which includes confidentiality, integrity, accountability, and availability. A registry key is security relevant if any of its containing name-value pairs are security relevant.

A project is initiated for the purpose of identifying all security relevant registry keys in Windows NT Registry. There are several motivations behind this project. First of all, some registry keys should be configured as protected resources which non-privileged user can not make arbitrary modification on. Usually, the decision about which registry keys should be protected comes either from specification, or developers' formal or informal documentations. As time goes on, however, the specification might become obsolete; it is hard to keep up with the evolution of software. Furthermore, people who made the decision regarding which registry keys should be protected might have left. So, from time to time, people might ask: "why is this registry key protected? what is the consequence if I do not protect it"? To answer these questions, software vendors have to turn to the developers, provided that the developers who made the decision are still there; otherwise they have to go through the specification and find out the result themselves. Specification could be obsolete and incomplete as well. yet nevertheless, compared with specification, program source code would provide more accurate, more complete and more up-to-date information. Therefore, if we can derive the security relevancy information from the program itself, especially if automatically, we can keep up with the evolution of the software regarding to the security relevant registry keys.

Secondly, various enterprise customers or developers from other groups want to know why a registry key is protected. These customers might want to build their own software on NT or port their software to NT. Sometimes, the software requires that a non-privileged user have the right to modify a certain registry key which is in the protection mode. They should either modify the software or remove the protection from the registry key. To make the right decision they would need to do risk analysis on whether it is appropriate to just remove the protection from the registry key. If the risk is not high enough, they might trade a little bit of security for the cost of modifying software. Usually customers are not satisfied with the specifications that only specify that a registry key should be protected without providing further details. The more details they have, the more accurate the risk analysis is.

Thirdly, the project hopes to identify security flaws related to the Registry. There still are several world-writable

registry keys after the Windows NT 4.0's fresh installation. Several NT security books [17, 10] have pointed out that some of the registries should be protected. We hope to identify the known one, as well as uncover the unknown ones if any.

3.2 Design and Implementation

Through the project we want to be able to answer the following questions:

1. Which registry keys/values are used in the program?
2. Where are they used?
3. Are they security relevant?
4. Why are they security relevant?

For the ease of implementation, we divide our task into two different steps. In the first step, we try to answer the first two questions by gleaning registry keys/values information from the program. The data itself is quite valuable, since it gives a global overview of the usage of the Registry by various components. For example, from the data we collected from Windows NT4.0 SP3 source codes, we found that Winlogon registry key is used 256 times throughout 33 different modules, and Lsa registry key is used 190 times throughout 24 different modules. This information suggests that we should be very cautious about changing the value, configuration, or the meaning of such registry keys. Fortunately, these two registry keys are protected in the default configuration and only Administrators and system can modify them. A data collection tool has been implemented for collecting the Registry usage information. Although it is impossible to resolve all the names of registry keys/values that are used in a program since some names of the keys/values are dynamically generated, we have indeed resolve 80% of them.

In the second step, the dependency and security relevancy analysis techniques discussed in section 2 are used for analyzing the security relevancy of each input from the Registry. Without the result from the first step, one can tell only whether an input from a registry key is security relevant or not without knowing in particular which registry key is security relevant. But when the first step and the second step results are combined, the security relevancy of a specific registry key/value is now ascertainable.

Data organization

The final results from the above two steps are stored in a database that contains the following fields:

- Registry key: this field records the name of a registry key used in a program, or whose value is used.
- Registry value name: if a registry value is retrieved, the field records the value name.
- Access permission for "Everyone" on this registry key: Since we are concerned with whether the key is world-readable or world-writable, only the permission for "Everyone" group (this group includes every user in the system) is recorded.
- Link to source file: this field provides a link to the source file that uses the registry key/value.
- Line number: this field records where in the source file the registry key/value is used.
- Security relevancy: the decision made as to whether the registry key/value is security relevant. The decision is based on the security relevancy analysis.
- Criterion: the reason of why the registry key/value is categorized as security relevant. Such reasons could be: the input is passed as a file name into a deletion function; or the input is passed as a file name into an execution function; or the input is used as condition to decide if a network connection function should be invoked, and so on.

Security relevancy analysis

Before security relevancy analysis is conducted, one question has to be answered: what consists of security critical action in the Windows NT operating system? The security action in the Windows NT is defined at system calls and library calls level, namely, system calls and library calls are categorized into two categories (security critical actions and security non-critical actions) based on the targets to which the actions are applied. Security critical actions in Windows NT are described in the following, and they are categorized by the targets to which the actions are applied:

- *Executable* : this kind of action usually involves executing a program, loading a DLL and executing its procedure, invoking a service and etc.
- *Permission or Privilege*: this kind of action usually involves setting or modifying a permission or a privilege on a target.
- *File or Directory*: this action involves accessing a file or a directory including reading, writing, and deleting.
- *Registry*: similar to accessing files, this kind of action only involves actions of accessing registry keys or values.

- *Network*: this kind of action involves accessing network, such as connecting, sending or receiving on network.
- *Environment Variable*: since a lot of other unexpected actions, whether security relevant or not, depend on environment variables, so a change to a security variable is considered security relevant.
- *Process and Service*: changing a process or a service is security critical, since an action might cause a denial-of-service problem if the target is inappropriate.
- *Security Policy*: Security policy, such as whether to allow somebody to login, is critical to system security, so any change to the security policy is considered security critical. However, in the Windows NT operating system, there is no standard API (Application Programming Interface) for this functionality. Sometimes, a policy is specified in a registry key, sometimes, it is specified in a file. It is very difficult to distinguish a normal file or a registry key accessing operation from the operations of accessing security policy. Our approach depends on manual annotation (either by programmers themselves or by code inspectors) to identify such an action.

Example

An example is used here to illustrate how the analysis technique presented in section 2 is applied to analyze security relevancy of registry keys/values. The program used in this example is the following:

```
f(){
  RegQueryKey(hkey, ... input)
  g(input);
}

g(char *str){
  char name[30];
  strcpy (name, "\\Winnt\\");
  strcat(name, str);
  h(name);
}

h(char *n){
  CreateProcess(n)
}
```

Figure 1 shows the dependency relationships among variables. Because `CreateProcess` is a security critical action, node $*n$ in the figure is marked as an S node, and because `RegQueryKey` is an input procedure, node $*input$ is marked as an I node. From the figure, a path from $*input$ node to $*n$ node exists, therefore, according to security relevancy analysis, input from `RegQueryKey()` is considered security relevant.

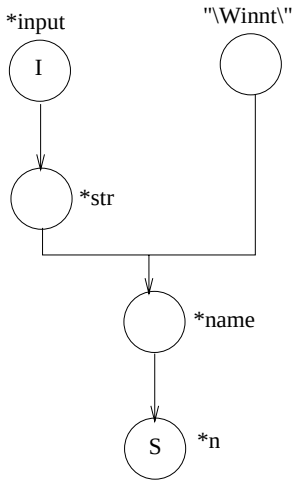

Figure 1: Data Dependency Graph

3.3 Results

We have applied the data collection tool on the whole Windows NT4.0 (SP3) source tree. There are 16,009 places where the Registry is accessed. The names of registry key/values used in 80% of those places have been resolved. The reason that the names of registry keys/values used in the other 20% have not been resolved is mainly because some registry key/value names are dynamically generated during the program execution; therefore, static analysis is impossible to resolve them.

Among all those 16,009 entries, 48% are just an "open" or "close" operation, which does not involve the real data exchange between the program and the Registry. 14% are "set" operations, which are considered as output as opposed to input. 25% are "query" operations which actually input data from the Registry to the program. The rest 13% consist of other registry operations which are of no interest to this project. Although "open", "close" and "set" operations sometimes have security impact on the system, they are beyond the scope of this work because we are only concerned about the security relevancy of the "input". For the "set" operation, if the output from this operation is never used as an input, the involved registry key is not security relevant; if the registry is used as an input somewhere, it will be under the category of "input", and will be analyzed in our approach. Therefore, in this project, only the registry keys involved in a "query" operation will be the target of our security relevancy analysis.

Based on the Registry usage information we have collected, experimental analysis using the technique discussed in section 2 was conducted on about 50 registry keys, 21 of which were found security relevant for various reasons. Among those security relevant registry keys, 11 are world-writable, which means if the program does not perform appropriate checks on the those inputs, an unprivileged user could be able to cause security breaches by modifying those registry keys. The rest of this section presents part of the results obtained from the analyses.

One of the interesting keys is `''HKLM\Software\Microsoft\Windows NT\CV\Type 1 Installer\Type 1 Fonts''` [1]. From the name of the registry key, it seems that this key contains information about fonts, which are unlikely to cause a serious security breach even if somebody can tamper with it. This is probably why the key is not protected. However, the analysis reveals that a `delete` action on files specified by this value. Therefore, if somebody makes the registry key point to an important file, this action will seriously affect the system.

`HKLM\Software\Microsoft\Windows NT\CV\ProfileList\(sid)` (sid is not a registry key name; it is a user's security ID and is user dependent) registry key contains a `ProfileImagePath` value, which is considered a directory name and will be appended with a string to form a file name. In a module executed in privileged context, this generated file name is passed onto a `delete` action, i.e. the file represented by this name will be deleted. If somebody can modify this value, such as making it point to other people's profile directory, the execution of this module will actually delete a undesired file, thus breaking system integrity.

The same registry key and same value are used to set the user's several environment variables. Considering that many applications may depend on those environment variables, a corruption of their values will lead to an undesired or, even worse, unsecured consequence.

`HKLM\Software\Microsoft\Windows NT\CV\Winlogon` registry key contains a `PolicyHandler` value. This value is treated as the name of a dynamic link library (`dll`) and a procedure name as well. The `dll` name is used to load the corresponding dynamic link library into the memory, and the procedure name is used to find the corresponding procedure from the loaded library to be invoked by the program. Thus, this value actually points to a piece of code, and compromising of this value will lead to the execution of an arbitrary code by the Winlogon module, which runs as a privileged process. Fortunately, this registry key is protected in the default configuration.

`HKLM\Software\Microsoft\NetDDE\Parameters\General` registry key contains a `DebugPath` registry value. This value contains the name of a log file. Our analysis result discloses that, in one of modules, the program conducts a `write` operation to the file specified by this registry value. A closer look at the program reveals that the programmer has not checked whether the registry value is trusted or not before going ahead to write to the specified file. Consequently, an implicit assumption is made by the programmer on the registry value. If the registry value is not protected, a malicious user can cause any file to be overwritten if that module is executed by a privileged user.

Knowing what registry keys/values are security relevant along with the permissions set on each of the registry key, it takes just a simple query on the database to find out all registry keys/values that are both security relevant and writable to "Everyone" group. Based on the result we have collected, we have identified 11 such registry keys/values, 4 of which are not documented in any literature that we are aware of. These results have been acknowledged by Microsoft Corporation.

As of the writing of this paper, only 50 registry keys have been analyzed at the initial stage of the analysis. We believe that with an analysis of all of the registry keys the number of unprotected security-relevant registry keys will be far more than 11. The results of this project are considered very useful by Windows NT security group, and thus are incorporated into the secure configuration of the Windows NT.

4 Related work

Several books [17, 10] have been published about NT security, most of which mention that some registry keys that should be protected are not protected in the default configuration Most of the suggestions come from analyses on windows NT operating system, from specification and documentation, or purely from experience. Ours analysis provides another perspective, which takes into consideration the final version of the source code. We therefore avoid the potential problems caused by inaccurate or obsolete documentation.

Static analysis technique has long been used as a technique to enhance program security. Although these studies are very similar in the way to deploy the technique, they deploy the technique to achieve different goals.

Bishop and Dilger studied one class of the time-of-check-to-time-of-use (TOCTTOU) flaws [1]. A TOCTTOU flaw occurs when an application checks for a particular characteristic of an object and then takes some action that assumes the characteristic still holds when in fact it does not. This approach focuses on a source-code based technique for identifying patterns of code which could have this programming condition flaw.

[1] For the sake of convenience, the following description uses `HKLM` to represent `HKEY_LOCAL_MACHINE`, and `CV` to represent `CurrentVersion`.

Fink and Levitt employ application-slicing technique to test privileged applications [5]. This static analysis technique is used for the program slicing according to the criteria derived from the specification. Orbek and Palsberg [14] have introduced trust analysis for high-order languages. Trust analysis encourages the programmer to make explicit the trustworthiness of data, and in return it can guarantee that no mistakes with respect to trust will be made at runtime. The similar static analysis technique is used in this paper to analyze the trustworthiness of data.

The difference between our work and those other works that uses static analysis technique in enhancing system security are the following: First of all, most of those techniques focus on detecting security violation, whereas our work focuses on pointing out the dependency relationship between inputs and the program's critical actions. While this dependency does not necessarily indicate a security vulnerability in the program, it reveals that as long as the input is not protected, or the input is not correctly checked, a security vulnerability is possible. This information may not lead to the discovery of a security vulnerability, but it indeed helps the testers look in the right place for the purpose of security testing; it also helps the developers make the right decision about whether or not to put extra efforts into validating an input. Secondly, some techniques require the modification of source code, such as annotating a source code. With the annotation of the code, analysis technique could collect more information from the code, thus leading to a more powerful analysis. However, given such a large system as the Windows NT, it is infeasible to modify the source codes before analysis.

Penetration testing [11, 15] is another way of discovering whether an input is security relevant or not by demonstrating that certain inputs could cause security breaches. In the case where the source code is not available this is an effective approach because all that needs to be done is to come up with a different input and feed it to the system to see whether the system security will be compromised? The disadvantage of this approach is that one has to see a security breaches to believe that an input is security relevant. If an execution path is never covered, it is difficult to determine whether the input related to that path is security relevant. In addition, devising a test case itself could be difficult.

5 Summary

We have argued and demonstrated that knowing the security relevancy of inputs is important to enhancing program security. In addition, we have presented a technique that reveals the security relevancy of an input. This technique is based on the insight that finding whether an input is security relevant is equivalent to finding the dependency relationship between the input and any security critical action.

We have also conducted experimental analyses on the Windows NT 4.0 source code. The results not only reveal the security relevancy information of registry keys/values, but also point out several vulnerabilities in the configuration of the Registry. These results demonstrate that security relevancy analysis is a useful technique in enhancing program security by pointing out the existing and potential vulnerability in the programs.

6 Acknowledgement

The authors are indebted to Peter Brundrett, Margaret Johnson, Kirk Soluk and other people in the Windows NT Security group for their insightful advice throughout the whole project. We are also grateful to Microsoft Corporation for providing us with the chance to conduct experimental analyses on the Windows NT 4.0 source code. We also thank the anonymous reviewers for their useful comments.

References

[1] M. Bishop and M. Dilger. Checking for race conditions in file acesses. *The USENIX Association Computing Systems*, 9(2):131–151, Spring 1996.

[2] J. Choi, M. Burke, P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *ACM-20th PoPL*, 1993.

[3] W. Chen and W. Berry. *Windows NT Registry Guide*. Addison-Wesley Developers Press, 1997.

[4] W. Du and A. Mathur. Vulnerability testing of software system using fault injection. Technical report, Purdue University, 1998.

[5] G. Fink and K. Levitt. Property-based testing of privileged programs. In *Proceedings of the 10th Annual Computer Security Applications Conference; Orlando, FL, USA; 1994 Dec 5-9*, 1994.

[6] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *POPL*, San Diego, CA USA, 1998.

[7] D. Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[8] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *SIGSOFT*, New Orleans, LA, USA, 1994.

[9] R. Kemmerer. Security, computer. In *Encyclopedia of Software Engineering*. 1994.

[10] N. Lambert and M. Patel. *PCWEEK Windows NT Security: System Administrator's Guide*. Ziff-Davis Press, 1997.

[11] R. R. Linde. Operating system penetration. In *AFIPS National Computer Conference*, pages pp. 361–368, 1975.

[12] A. Diwan, K. S. McKinley and J. B. Moss. Type-based alias analysis. In *SIGPLAN*, Montreal, Canada, 1998.

[13] J. Ferrante, K. J. Ottenstein and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.

[14] J. Palsberg, and P. Orbek. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, pages 314–329, September 1995.

[15] C. Pfleeger, S. Pfleeger and M. Theofanos. A methodology for penetration testing. *Computers and Security*, 8(7):613–620, 1989.

[16] S. Horwitz, T. Reps and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[17] C. Rutstein. *Guide to Windows NT Security: A Practical Guide to Securing Windows NT Servers & Workstations*. McGraw-Hill, 1997.

[18] A. Aho, R. Sethi and J. D. Ulman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[19] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL*, Paris, France, 1997.

[20] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, St. Petersburg FLA, 1996.