The User is Not the Enemy: Fighting Malware by Tracking User Intentions

Jeffrey Shirley
University of Virginia
Charlottesville, Virginia

jshirley@cs.virginia.edu

David Evans
University of Virginia
Charlottesville, Virginia

evans@cs.virginia.edu

ABSTRACT

Current access control policies provide no mechanisms for incorporating user behavior in access control decisions, even though the way a user interacts with a program often indicates what the user expects that program to do. We develop a new approach to access control, focusing on single-user systems, in which the complete history of user and program actions can be used to improve the precision and expressiveness of access control policies. We describe mechanisms for securely capturing user actions, mapping those actions onto likely user intents, and a language for defining access control policies that incorporate user intentions. We implemented a prototype for capturing user intentions, and present results from experiments on malware mitigation using the prototype. Our results show that a very simple MAC policy can prevent a significant amount of system damage caused by malware while not interfering with most benign software.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection – *access controls, invasive software*.

General Terms

Security

Keywords

User intent, access control, security policies.

1. INTRODUCTION

Access control mechanisms are a powerful tool for controlling actions taken by software, but an effective and usable means for securing desktop applications and operating system software remains elusive. The purpose of access control mechanisms in a single-user desktop environment should not be primarily to place restrictions on actions taken by users, but instead to limit the permissions granted to particular processes at specific times. Since the user of a single-user system is its owner, security threats are not caused by malicious users but instead by exploitable, flawed, or malicious software. Hence, security mechanisms should be designed from the perspective that the user is a useful partner in making good security decisions, rather than the more traditional perspective in which a possibly malicious user is the enemy.

Existing policy-based access control implementations have not been practical enough for widespread use in single-user environments. As a result, applications tend to either be run completely unsecured (beyond operating system discretionary access control mechanisms) or in a sandboxed environment that significantly limits the utility of the application. Restrictive mandatory access control policies have generally been seen as too limiting or completely unusable for use in the single-user environment. Application-specific policies often suffer because it is impractical to expect users to create policies for individual applications and because administrator or developer-deployed policies may not be able to meet all user needs.

Current approaches fail to take advantage of a valuable source of information, namely, the way the user installs, launches, and interacts with programs. Although user actions are inherently ambiguous, they provide strong indications of how much a user trusts a program and what a user believes a program should do. We propose to augment the current approach where access control decisions are made based on a static system configuration and policy, with a new approach

NSPW'08, September 22–25, 2008, Lake Tahoe, California, USA.

where decisions can be made based on the entire lifetime of a system, including user interactions and program actions. Dynamic logs of user interactions and program executions can provide information not available to the author of a static policy such as filenames, host names, and other data entered via the user interface.

Contributions. In this paper, we present a design for securely capturing and analyzing user and program actions, and explore the possibility of taking advantage of this valuable information to enable more secure access control for desktop applications. Incorporating user actions in policy rules enables mandatory access control policies that are both simple and powerful. When applied to all running programs, these policies can prevent damage caused by malware by using user intent information to identify benign program actions. As an example, we provide a policy that allows programs to access files based on the user's natural interactions with the program's user interface. User intent knowledge enables the policy to distinguish between benign and malicious file access. We describe a prototype implementation for Windows and present results on using it to limit file access by malware and benign program executions.

Previous work. Cui first proposed an intrusion detection mechanism combining information about user intentions (from GUI events) with network intrusion detection [8]. In the BINDER IDS, the presence of a user input event, such as a mouse click, was correlated to outgoing network traffic. The presence of network traffic not connected temporally to GUI events was used as a heuristic for the presence of malicious traffic outgoing from a host. Cui's results demonstrate that user intent can increase the accuracy of network IDS.

The Polaris [33] access control system developed a method of constraining Windows programs that can indirectly take some user intent information into account when making security decisions. Polaris runs programs under a specially modified user account with only a more limited set of permissions needed for the program to function. Polaris also supports file designation, meaning that users can implicitly designate file permissions when using an application. Polaris replaces the standard file chooser dialog box with a special file chooser that can add specific file permissions to the underlying set of application permissions. The Polaris system illustrates the potential value of using user interactions in access control, but does not apply the idea more generally and does not make user interactions available for use in explicitly configurable security policies. We provide a general mechanism for incorporating user interactions into access control policies in ways that enable richer security policies.

Roadmap. Section 2 discusses our system design and prototype implementation of the policy engine. Section 3 describes experiments analyzing the effectiveness of our intent-based anti-malware policies. Section 4 describes the effects of applying intent-based policies to benign programs. We describe other related work in Section 5, and conclude with discussion in Section 6.

2. DESIGN

A user-intent-based access control (UIBAC) system involves three basic components, shown in Figure 1. The first component collects and records user interface events (Section 2.1). The log of user behavior is used as input to the user intention inference component (Section 2.2). Finally, inferred user intent concepts generated by the inference component are used in intent-based security policies in order to make enforcement decisions (Section 2.3). Our focus is on protecting file resources from damage caused by malware or by exploits against vulnerable software.

2.1 Prototype

We built a prototype system incorporating three components: a collection mechanism, an inference engine, and policies and enforcement mechanisms. The purpose of the prototype implementation is to evaluate the potential for intent-based access control policies by determining if policies can be found that allow benign applications to (mostly) execute normally, while blocking malicious activities.

Since our goal is to explore possible policies and measure their effectiveness, the prototype was not constructed to resist targeted attempts to circumvent or corrupt the system, although the component designs in the following sections suggest how a secure implementation could be built. We implemented event collection for Win32 applications. Events describing user interaction with the environment are fed into a common logging system, along with system resource events such as file, network, and registry accesses. The intention inference component scans this common log for patterns matching user intent concepts referenced in the intent-based security policies themselves. The policies control access to protected resources. For the experiments described in this paper, we performed the intention inference and policy checking components using an offline analysis on the logs collected by the

event collection component. This is sufficient to answer our main research question: *does knowledge of user intent enable MAC policies that protect against malware without unacceptably disrupting benign programs?* In the following sections, we present a secure design, and describe the actual prototype implementation.

2.2 Event Collection

We need a trustworthy method for recording user actions as well as program actions since this information will be used to make access control decisions. Activity information includes both concrete user actions, such as mouse clicks or keystrokes entered, and contextual information describing the state of the user interface at the time the user activity took place, such as active window and button text. A diverse selection of user interface toolkits and libraries is available to application developers, including some with unique features. The ability to log events from multiple toolkits may be required to capture some forms of user intent, but most toolkits ultimately are implemented using the native Windows API. All of this information needs to be gathered in a secure way if we are to derive trust from inferences made on the recorded activity. It should not be possible for a malicious application to inject fabricated events into the collection mechanism in an undetected manner, to interact with a user in a way that is not observed by the collection mechanism, or to corrupt or obscure real user interface events or context observed by the collection mechanism.

One way to build a secure collection mechanism would be to use a virtual machine, which can observe user actions and program behaviors from outside the guest OS. If virtual machines are used as part of the access controller to isolate applications, the input events can be compared at the level of the host OS to the input events taking place within the isolated guest OS to provide input validation without requiring a major redesign of programs or operating systems. Assuming the VM provides good isolation, this design ensures that all user interactions with isolated applications are observed by the collection component. In order to protect the integrity of the context data, the VM would also need to monitor certain files and user interface resources. Such an approach might be built on top of the Virtual Machine Communication Interface API [42] for VMWare, for example. There are also software-based solutions to this problem in development by the major operating system vendors as part of the push towards trusted computing [44]. Future operating systems may be designed to protect the integrity of user interfaces, since their protection has serious implications for security.

Prototype Implementation. For our prototype, we developed a simple event collector and logger for Win32 applications. The logger also records program behavior (as opposed to user behavior) when programs access restricted resources (files, network resources, and the Windows registry).

Building an event collector for Win32 is difficult since events are visible only at a relatively low level. Our prototype uses an API hooking technique (based on the Detours system from Microsoft Research [18]) to obtain various user interface and program behavior events.

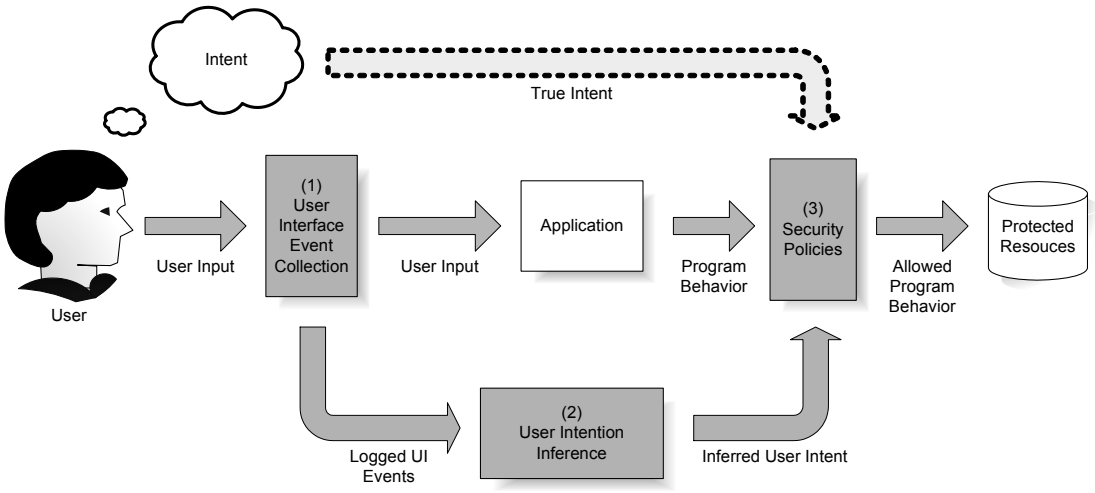

Figure 1. User-Intent-Based Access Control

Hooks are inserted into monitored processes via a DLL loaded into all running process' respective address spaces. The DLL contains a callback method that watches for specific monitored user interface events generated by the program's user interface. The hook callback has access to all of the monitored process' state, since it runs in-process. When a monitored event is observed, the callback function gathers any ancillary information (such as button identifiers and captions) needed by the logger to record the context of the event, either directly from the parameters of calls to a Win32 API function or by using the parameters (which often include file and window handles, etc.) to query for the necessary information. This information is then written to a common log file along with a timestamp, the process identifier, and the name of the program generating the event.

Our Win32 prototype is suitable for conducting experiments on the usefulness of UIBAC policies, even though it has several limitations that would preclude its use in a practical system. It is not designed to be difficult to circumvent; a targeted attack could easily be designed to interfere with the local hooks in memory. Some potential concurrency issues also remain in the prototype implementation. Events are recorded in the order in which the hooked callback functions are executed, with a recorded millisecond-granularity timestamp being generated at that time. Although overlap is possible due to the presence of separate buffers for each individual process writing to the shared log file, this has not yet been a problem in our testing and experiments. The exact ordering of low-level program behaviors from different programs seems to rarely be of critical importance.

2.3 User Intent Inference

The second component needed by our intent-based access controller is an algorithm for inferring user intents from the log of user interface events. Intent concepts provide an abstraction over the concrete user behavior. We wish to express policies at the level of intent concepts, not specific events, since most events are at too low a level for understandable policies. Further, there may be many different ways to convey the same intent concept. For example, when a user selects and opens a file in a file chooser dialog, a sequence of events takes place as the dialog is created, the user searches for a file, and finally opens the file by clicking on a specific button or using the keyboard. The intent inference algorithm interprets this series of events as the concept "user intends to open file <*pathname*>". We formalize this concept as,

file_open(Filename *f* , Process *p*)

The user, interacting with process *p*, intends to open file *f*.

The context information in this case is important in understanding what file access permissions (e.g., read or write) the user intended to grant to the program presenting the dialog box. More than one pattern of concrete user behavior may be mapped to a single user intent concept. Users may express their intent to open a file in a program in multiple ways, such as by dragging and dropping file icons in Windows, using dialog boxes, and using recent document menus. A sophisticated intent inference engine would be able to infer the same intent, regardless of the many different ways a user may express a particular intention.

When a security decision needs to be made during policy enforcement, the inference engine may be called upon by an intent-based security policy to check whether a particular user intent concept is present and applicable to the program attempting a restricted action. The inference engine then checks to see whether that intent concept is implied for the subject process by the user behavior log. The log is kept indefinitely, so the policy has the capability to infer user intent dating back to the initial installation of the system[1]. This opens up the possibility of inferring all configuration changes made to the program via the user interface, including those made during the installation itself.

For example, if a security decision is needed for a particular file that the program would not otherwise have access to, an access controller enforcing an intent-based policy will check the log to see if intent to open that file can be inferred from the recorded events. This intent could be expressed through the series of user input events corresponding to creation of the file chooser dialog and selection of the file. Although we cannot know the actual user intent, we can infer that intent to open the file is a possible interpretation of the sequence of events. We infer an intent concept if a particular intent can be expressed as the observed sequence of events.

[1] Machines may ship with a preloaded events log to reflect programs installed by the vendor using a pre-built image. To apply user-action based policies to those programs, it may be useful to supply an initial log that contains the minimal actions a user would have done to install the preloaded programs.

In addition to user intent concepts, the inference engine also infers concepts based on program actions. For example, if a new file is created by a given program, the concept,

program_creates_file(Filename *f*, Executable *e*)

is inferred where *e* corresponds to the executable whose instantiation created the file.

Appendix A provides a full list of the intention concepts inferred by our prototype implementation.

Prototype implementation. Our prototype is adequate for the simple concepts examined for this paper. Future work will investigate whether a more sophisticated algorithm is needed and useful for inferring other interesting intent concepts (for example, concepts that cannot be expressed as regular expressions or concepts that depend on temporal notions).

Our prototype implementation maps concrete events recorded in the log to user intent concepts. It does this by looking for patterns in the sequence of user interactions and program behavior that correspond to regular expressions representing user intent concepts. Concepts are inferred based on two components: the general pattern being followed, such as file selection (specified as part of the inference engine) and the context (specified as part of the UIBAC policy).

The algorithm searches the log and looks for an event matching the starting point of the regular expression for a particular inference pattern. It continues through the general inference pattern, checking against the log to determine whether the pattern is satisfied. Context information from the user interface can be bound to variables in the general pattern. If the policy does not specify a concrete value for a variable, it is bound when it is first encountered in the log. The substitution is performed using regular expressions with grouping.

Our prototype can also infer program behavior patterns derived purely from the dynamic program behavior recorded in the log. Such dynamic behavior inferences can still be useful in a security policy.

2.4 User Intent-Based Security Policies

The third component of our system is the policy enforcement mechanisms and the intent-based security policies themselves. The intent-based policies provide a mapping between inferred user intent concepts and behavior patterns and permissions they imply. These permissions are similar to those in traditional access control work, and designed to mediate access to resources such as files, network connections, and execution permissions. Our system can observe many system configuration changes by observing user behavior and intent, and thus can create much more flexible and durable policies than is possible under many other access control systems.

In this work, we focus on finding policies capable of enhancing security for typical desktop computers. We seek a MAC policy that can suppress classes of unwanted behavior without disrupting the function of benign applications. Since such a MAC policy can be applied to all processes, it can prevent undetected malware from doing damage to the system. We describe some experiments with various candidate MAC policies in Section 4. A successful MAC policy would prevent damage to system resources while still allowing most programs to function normally.

Implementation. Our policy interpreter uses a policy language based loosely on the Java security policy language. Rules in the policies map user intent concepts to changes to the set of permissions granted to that process at any given time. Policy GRANT rules add permissions to the active permissions possessed by a process, which may be specified by executable image name (e.g., *firefox.exe*) or PID (e.g., 4526). We may also add additional flexibility in the future so that permissions can be finer grained for dynamically loaded program modules and interpreters executing at a higher level within the application.

An access controller enforcing an intent-based policy mediates attempts to access protected resources. If a requested permission is implied by the set of permissions for the process making the request, the action is allowed; otherwise, it is denied. Our prototype policy engine allows us to test various intent-based policies against logs of user behavior collected during experiments, but is not yet embedded into an access controller. Instead, policy violations are simply flagged for analysis, since our focus for this paper was to evaluate intent-based anti-malware policies.

2.5 Policy Language

The general form of policy rules specifying intent-based dynamic policies in our language is:

intent (*context*) GRANTS (*subject*) *permission*

The left side of a rule identifies an intention inferred from user actions, as described in Section 2.3. The name of the intention concept is *intent* (e.g., file_open), and the context can include bound and unbound variables. Context variable types include Filename, Process, Executable, and Host. Variables

are bound to values derived from the log information and may be used as subjects or as part of a permission grant.

The *subject* refers to the entity to which the permission is being granted. We currently support the principals Executable and Process. The Executable principal grants the permission to all instances of a program, while the Process principal applies the permission only to a specific process.[2] Since our logs may carry information collected from many instances of the program, this is an important distinction.

The *permission* indicates the access being granted, such as Read or Write; permissions can use variables that were bound in the *context* information.

As an example, this rule grants processes access to write a file following a user intention to open the file:

file_open (Filename f, Process p) GRANTS
(Process p) Write (Filename f)

This rule uses the file_open intent concept. The two context variables on the left-hand-side of the rule are bound by the inference engine during evaluation of the rule to a specific filename specified by the user in process p's user interface. These same variables are then used on the right-hand-side to limit the scope of the permission grant to only the same process and file. A single policy rule such as this can match multiple times with different variable bindings.

3. CANDIDATE RULES

In order to examine the effectiveness of intent-based policies, we developed a set of candidate MAC policy rules. These rules assume a system-wide default-deny policy for writes to existing files, with a few exceptions such as the Windows registry files and temporary directory. The purpose of the policy rules is to loosen this default MAC policy so that benign programs can still function, but not so much that a large amount of damage by malware can take place. We do not attempt to provide any confidentiality protection against malware (hence, there are no restrictions on file reads). Each program is given control over files created by it or installed with it, as well as files that the user directly intended to allow the program to access. We do not claim to provide comprehensive MAC policies at this stage of development; our experiments were instead designed to explore some of the issues related to designing such a policy using intent-based rules. Next, we describe the intuition behind each of the candidate policy rules.

The *INSTALLED-FILES* rule states that if a user intends to install a program, then the program can write to any of the files created by the installation:

INSTALLED-FILES:
program_installation (Filename *f*, Executable *e*)
GRANTS (Executable *e*) Write (Filename *f*)

The variable *e* is bound by the inference engine to all executable files that are part of the installation itself (not the installer program). The inference pattern in question looks for a user-initiated installation process by looking for the creation of dialog boxes characteristic of installer programs, and binds any files written to by the installation process to the variable *f*, until the installation program ends.

The way we identify program installations is certainly not secure against malicious programs; a secure implementation would require additional controls on the installation process, such as rules governing the ability to initiate the install process. We are exploring ways of integrating our installation rules more closely with the operating system in order to make them more secure and reliable.

This rule could be implemented as a static grant of permission to files created during program installation, but this would require manual creation of a policy controlling the specific files as opposed to the automated process that the intent-based policies allow. The ability to recognize the specific user intention to grant install privileges to the installation program makes it unnecessary to manually designate which programs are installers.

The *CREATED-FILES* rule is very similar to the *INSTALLED-FILES* rule, except that it concerns files created by the program itself:

CREATED-FILES:
program_creates_file (Filename *f*, Executable *e*)
GRANTS (Executable *e*) Write (Filename *f*)

This rule allows programs to write to any files they create; this is useful for allowing programs to create temporary files and caches. This rule is not a user intent rule but a program behavior rule. Files created by a program remain associated with the program for integrity purposes. Note that this rule permits

[2] In order to prevent name conflicts caused by process ID reuse, the inference engine also logs whenever a new executable process is started.

malicious file-dropping behavior; a more precise rule would place limits on the types and locations of files that an executable can create.

The final two rules grant a program permission to write to files that the user specifies via the user interface:

SAVED-FILES:
file_save (Filename *f*, Process *p*) GRANTS
(Process *p*) Write (Filename *f*)

OPENED-FILES:
file_open (Filename *f*, Process *p*) GRANTS
(Process *p*) Write (Filename *f*)

The file_save and file_open intent concepts are inferred from user interactions with the program's user interface. These candidate rules do not distinguish between opening a file for writing and opening a file for reading only; a more precise policy could take advantage of these distinctions to provide finer-grain control of permissions.

One might wonder why the *SAVED-FILES* rule is necessary in light of the *CREATED-FILES* rule. The reason is that sometimes users will specify that a saved file replace an existing file created by another program. Likewise, when a user intends to open a file created by another program, the user will often need to save to it using the new program later.

In the next section, we describe some preliminary experiments we conducted to measure the effectiveness of these rules and to determine what combination of rules best allows benign programs to function properly while still providing integrity protection against malware.

4. PRELIMINARY EXPERIMENTS

To find MAC policies that achieve our goal of protecting file resources from damage caused by malware, we conduct three kinds of experiments: *malware protection* experiments where we run known malware and evaluate how much of its malicious behaviors would be detected by our candidate policy rules; *deployment* experiments in which we test our candidate policies on a Windows desktop system in normal use to see how frequently a benign program violates candidate rules; and *benign program* experiments, in which we perform a series of artificial tests on Windows software to check for false positive violations of the policy.

All our policy rules grant permissions rather than taking them away; thus, they loosen the permissions applied to a program. Programs have no write permissions for existing files until they are specifically granted to them by a matched rule, but we allow all programs read-only access to all files since our focus is on protecting file integrity, rather than confidentiality. The best MAC policy, in accordance with the principle of least authority, is one that grants the least permission while still allowing benign programs to function. We intend to test policies designed to provide confidentiality in the future; for these tests, we would also use a default-deny policy for reads of existing files.

4.1 Malware Detection Experiment

Our first experiment was intended to determine whether the integrity of programs and files is still protected from malware when various intent-based permission loosening rules, described in Section 3, are in place (note that a policy with no rules would not allow any file writes, so would provide file integrity protection against malware, but would also not allow any useful file-modifying behaviors). For this experiment we did not attempt to block malware from executing altogether or to prevent it from residing in memory. We were only concerned with file integrity and whether it could still be preserved with various rules in place.

Malware selection. We tested a set of malware programs taken from the offensivecomputing.net online repository collection point, representing the most prevalent reported malware as of early 2008 and a number of well-known worms (sometimes with multiple samples and variants of each tested). We also included a set of unknown malware samples collected on a honeypot by Christopher Kruegel. These were samples found to have modified files on the honeypot.

We excluded samples that were not found to cause any damage to file resources in the test environment. Possible causes of sample exclusion included malware samples that failed to execute, malware that executed but did not attempt to damage file resources due to the short duration of the tests, and malware that executed but did not attempt to damage file resources due to the specific test environment used (for example, the operating system version, installed programs, or the presence of a virtual machine). Any malware that altered or created new files was included.

Test procedure. Malware was executed on a VMWare-based testbed. A Windows XP SP2 installation was set up in a virtual machine, with a number of benign programs installed and the latest patches from Microsoft. We used an automated script to inject

malware samples into the guest operating system and execute them with our intent-based logging component running. There was no human interaction with the VM during the tests. Each malware sample was allowed to execute for 8 minutes. At the conclusion of the test, the VM was shut down, the log file was collected, and the disk image was checked for evidence of damage to file resources. This check was carried out using MD5 checksums of the files in the VM disk image; normal file changes performed by the OS were excluded from the definition of file damage. This left 28 samples, which we used in our experiments.

For the malware samples confirmed to cause damage to files on the VM, our policy checking tool was run on the collected logs in order to determine whether the damage would still have taken place each of the permission granting rules describe in the previous section (*INSTALLED-FILES*, *CREATED-FILES*, *SAVED-FILES*, *OPENED-FILES*). Since malware often attempts to damage multiple files, it is possible for a rule to provide only partial protection by allowing some damaging actions by not others. We have tested malware with two slightly different default-deny policies. The first of these default deny policies denied all file modifications of existing files, but allowed the creation of new files. The second denied all file writes, including the creation of new files.

Results. All of the samples except for Bagle attempted to write to an existing file. This would be disallowed by the draconian default deny policy. All of the file writes observed in our testing would also have been disallowed with all of the permission-granting rules in place. This is not a surprising result, since the virus is installed surreptitiously (thus it never acquires any of the installation privileges), and the user never interacts explicitly with the virus process. One malware sample, with an identity unknown to commercial virus scanners, was able to modify files without being stopped by the policies. It appears that this malware used a kernel level rootkit combined with privilege escalation to accomplish this. As we have not yet attempted to harden our system against circumvention, this is not a significant negative result. In any case, our prototype implementation cannot provide access control once the kernel is compromised.

Under the more restrictive default-deny-write policy, prohibiting the creation of new files, we again observed that intent-based policy rules did not allow the malware to modify the filesystem significantly more so than the default-deny policy. The one exception was the same malware sample mentioned above that was able to create files without being detected.

Windows Registry settings used to execute code at a later date are likely in use for some of this malware. Our logger can observe registry activity but we are still experimenting with the best possible intent-based policies for controlling registry access and thus the execution of this type of malware. Registry integrity policies will likely be quite similar to file integrity policies.

The intent-based rules do not seem to significantly reduce the security provided by the two disallow all policies as none of the malware samples attempted operations that caused any of the intent-based rules to be triggered. However, this test is limited in scope and might not include enough samples or a broad enough definition of damage. In addition, this policy could easily be evaded by malware designed to evade it. Nonetheless, these preliminary experiments provide some support for the hypothesis that a restrictive file modification policy can prevent some malicious behaviors, and loosening it with the kinds of UIBAC rules we propose does not diminish its effectiveness, while allowing benign programs to perform more actions, as discussed in the next section.

4.2 Deployment Experiment

In order to determine whether a useful MAC policy can be devised, we also performed a test under presumably benign conditions. The goal of this experiment was to determine whether these policy rules would allow benign programs to function normally. During this experiment, the logger was allowed to run normally for a day on the machines of two users (one technical and one non-technical). A total of 60,541 file-related events were recorded. An event was recorded whenever a program opened a file for reading, writing, or deletion. Policies composed of combinations of the rules tested against malware in Section 3.1 were evaluated against these logs to determine whether or not they would interfere with the functioning of legitimate Windows software.

A limitation of the test was that unlike in our intended model of use, the logger was not in place and collecting data throughout the entire installed history of the users' machine. This made the installation policy rule and program creation rules more difficult to evaluate. In order to simulate this part of the data, we simply augmented the logs with a logged installation and first execution of the programs of interest to determine which files would have been allowed under

these rules. In general this amounted to only a few directories (such as the program's installation location and sometimes a directory used to store state information). However, there may be some mismatch between directories in the augmented logs and the actual installations under observation.

Results. Table 1 summarizes the results from our deployment test. With a few notable exceptions, benign programs generally still functioned with all of the candidate rules in place, with a relatively small number of false positives. Most of the writes allowed by our rules were enabled by the *CREATED-FILES*, reflecting the fact that most file writes occur to files created by the programs themselves.

In our tests, the *INSTALLED-FILES* rule accounted for just two of the file writes that would have otherwise been denied. This is perhaps not too surprising if benign programs only seldom write to their own installation files. In our experiment, most of the files were created by the programs themselves rather than the installers. This could also be a limitation of our test procedures, since only a few programs were observed to be in use during the test.

The *SAVED-FILES* and *OPENED-FILES* rules allowed 16 additional file writes to occur. While this number might seem small, this includes only files explicitly opened by users in a user interface. Thus, it generally only includes files that contain user-generated data. While the number of writes to these files is lower, these writes might be considered among the most important. We would expect to observe more in a longer deployment test.

False positives remaining included several situations where a program wrote data to a file created by a separate program. One of these concerned an external program that is bundled as a plugin with many of the Mozilla programs. This program was bundled with some of the Mozilla programs, but not reinstalled when a second Mozilla program was installed during our test installation. Thus, the second program did not have access under the installation rule to the external program's files. This might be addressed using a more intelligent install rule, or by allowing policies to assign permissions to plugin components separately from the main program. A similar situation was observed in our artificial benign program experiments with a component designed to embed the Internet Explorer web browser into other programs.

Table 1. Deployment Experiment Results. Policy violations encountered over all 60541 file access events.

Policy	Violations
Default-Deny-Existing Policy Only	11050
INSTALLED-FILES	11048
CREATED-FILES	49
INSTALLED-FILES+*CREATED-FILES*	47
SAVED-FILES+*OPENED-FILES*	11034
INSTALLED-FILES+*CREATED-FILES*+ *SAVED-FILES*+*OPENED-FILES*	31

There were several other false positives that appear to be the result of limitations in the test procedures (the lack of a complete history of all the files created by a program in the past) or engineering limitations of our logging mechanism. In the latter category a number of files that appear to have opened by users were missed by the file open/file save rules. A possible explanation for this is that the users used an unsupported means of expressing their intent to open or save a file (such as the Recent Documents menu). Because the inference component is separate from the policy, we expect that this problem can be solved with more complete support for the Windows toolkit in the inference patterns.

4.3 Benign Program Experiment

Because we felt that the limited deployment experiment described in Section 4.2 gave an incomplete picture of the potential for false positives, we also tested our policies in a series of contrived tests on a collection of 46 benign programs. In these tests, we installed and executed the benign programs for a series of 8 minutes, with a script to select various menu options at random. We then checked to see if any false positives were observed for our policies during the tests. The programs and the results of this testing are listed in Appendix B.

Results. In our contrived tests, we observed that 34 of the benign programs did not create any false positives. Eleven of the sample programs created false positives only in executing an embedded Internet Explorer plugin component that modified the IE cookie database and history files. Many programs use Internet Explorer in order to add web functionality, and the embedding of the plugin caused writes to the IE history and cookie databases to appear to come from the containing programs rather than IE itself. This problem could potentially be resolved in a number of ways. We could simply consider the IE database files a shared part of the operating system and allow all programs to alter it; this would probably be relatively safe, although some potential would exist for programs to destroy data stored in cookies by other programs. We might also

add support for the IE user interface elements to our inference patterns so that programs could be granted permission to use the IE component by a policy recognizing it as a standard Windows interface type. Finally, we could add support for granting permissions to specific modules or libraries loaded into programs in addition to the programs themselves.

One additional program, a spyware scanner, deleted a number of files in violation of the policy. We would also expect virus scanners to exhibit similar behavior, though we did not include one in our tests. Windows does provide support for registering these security-related programs in the Registry, and could exploit that process to give special permission for file removal or alteration in these cases. There is also some user interaction before files are actually deleted, so more sophisticated UIBAC policies could potentially account for this situation.

5. RELATED WORK

We mentioned the most closely related work in the introduction; here, we survey relevant work in several related areas: usability and security, user intent, security policies and access control, intrusion detection, and threats posed by programs.

Making security mechanisms usable (from an HCI perspective) has been a difficult task for the security community, though it is frequently recognized as an important goal [1, 7, 40]. Several authors have argued that security mechanisms are inherently and particularly difficult to make usable. Yee recognized that user actions might implicitly designate security decisions for an application [46]. Our work attempts to leverage the effort put into application usability to assist with security decision-making. Instead of expecting users to explicitly define security policies or deal with security interruptions, we aim to define security policies implicitly based on the actions users are taking normally.

User intention concepts provide an abstraction in our system over policies based purely on user interaction events, and provide a flexibility not present in systems like BINDER or Polaris that use user behaviors directly. Policy languages themselves typically fall into two main types: those based on independent evaluation of policy rules [35, 25, 28] and those incorporating more complex and powerful constructs such as first-order logic [30]. There are tradeoffs involved with each approach, though the former has been more popular because the policies are typically easier to write and reason about [39]. Our prototype system uses a very simple language of the former type.

Mandatory access control mechanisms, coupled with strict security policies, have the potential to prevent many security problems caused by malicious and insecure programs. Several access control systems of this type have been developed, including notably SELinux [25] and AppArmor [27]. Our mechanism differs in that it is not intended only to detect application anomalies but also to prevent applications from functioning in ways that deviate from user intentions. This includes program anomalies caused by attacks against software vulnerabilities, but also a larger class of behavior that is simply unwanted by users, such as when a program deletes files unexpectedly. Much previous work has attempted to detect or prevent security threats to desktop operating systems and software by checking for attack behavior or violations of security policies. Signature-based systems [37] have attempted to detect attacks on desktop security primarily by checking for exploitation of known vulnerabilities, but suffer from the inability to detect attacks against unknown vulnerabilities. Non-signature-based or hybrid systems, such as those using anomaly-based intrusion detection or behavioral heuristics [5, 17, 19, 38, 29, 36] have often suffered from a high false positive rate causing them to be impractical for real-world use [14]. These systems may also be vulnerable to mimicry attacks [43], while our system can provide some measure of protection against this threat if we can secure the input path. Our system differs from purely behavioral systems in that it takes advantage of the history of user and program actions.

6. CONCLUSION

Our approach potentially offers a number of advantages over traditional access control. First, it could allow for easier policy development. Since intent-based policies reference higher-level abstractions of intent rather than lower-level application behavior, they can be more readily comprehensible to humans and amenable to automated policy development. Second, intent-based policies can promote greater reusability of policies, since they do not depend on the specific details of how individual applications carry out that intent. Third, intent-based policies align well with usability, since they do not require extensive user configuration or decision-making other than through normal application usage. Finally, they offer a good opportunity to dynamically track changes in application configuration over the entire installed lifetime of a program. Many configuration changes are observable via user

behavior, so policies can adapt over time to match changes in application behavior associated with program configuration changes.

Our experiments are not yet enough to demonstrate the effectiveness of our approach against sophisticated malware, but they indicate that adding support for UIBAC rules enables stricter behavior restrictions which can limit the damage malware can perform. Our preliminary experiments with benign programs provide some hope that policies can be designed that allow nearly all well-designed desktop applications to function normally.

ACKNOWLEDGEMENTS

The authors would like to thank Chris Kruegel for generously providing us with a set of malware samples for our experiments. We are also grateful for the feedback provided by the anonymous reviewers and attendees at NSPW.

This work was supported in part by funding from the National Science Foundation (grants 0541123 and 0627527).

REFERENCES

[1] Anne Adams, and Martina Angela Sasse. Users Are Not the Enemy. Communications of the ACM, December 1999: 40-46.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proc*. *Symposium on Operating System Principles.* 2003.

[3] D. Elliot Bell and Leonard J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report. The MITRE Corporation, 1973.

[4] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report. The MITRE Corporation, 1977.

[5] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-Aware Malware Detection. In *Proc. IEEE Symposium on Security and Privacy.* 2005.

[6] David D. Clark and David D. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proc. IEEE Symposium on Security and Privacy.* 1987.

[7] Lorrie Cranor, and Simson Garfinkel. *Security and Usability.* O'Reilly, 2005.

[8] Weidong Cui, Randy H. Katz, and Wai-tian Tan**.** BINDER: An Extrusion-based Break-In Detector for Personal Computers. In *Proc. USENIX Security Symposium.* 2005.

[9] T. Daboczi, I. Kollar, G. Simon, and T. Megyeri. How to test graphical user interfaces. *IEEE Instrumentation & Measurement Magazine*, September 2003: 27-33.

[10] Dorothy E. Denning. A Lattice Model of Secure Information Flow. Communications of the ACM, May 1976: 236-243.

[11] Rachna Dhamija, J.D. Tygar, and Marti Hearst. Why Phishing Works. In *Proc. ACM SIGCHI.* 2006.

[12] DoD Standard 5200.28-STD: Trusted Computer System Evaluation Criteria. United States Department of Defense, 1985.

[13] David Ferraiolo, and Richard Kuhn. Role-based Access Control. In *Proc. National Computer Security Conference*. 1992.

[14] Carrie Gates and Carol Taylor. Challenging the Anomaly Detection Paradigm: A Provocative Discussion. In *Proc. New Security Paradigms Workshop.* 2006.

[15] Joseph Halpern, and Vicky Weissman. Using first-order logic to reason about policies. In *Computer Security Foundations Workshop.* 2003.

[16] Steven B. Hirsch. Secure Keyboard Input Terminal. U.S. Patent 4,333,090. 1980.

[17] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 1998: 151-180.

[18] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proc.USENIX Windows NT Symposium*.1999.

[19] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based Spyware Detection. In *Proc. USENIX Security Symposium.* 2006.

[20] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proc. Annual Computer Security Applications Conference.* 2004.

[21] Henry M. Levy. *Capability-based Computer Systems.* Digital Press, 1984.

[22] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, and Ruth C. Taylor. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computer Systems. In *Proc. National Information Systems Security Conference.* 1998.

[23] Microsoft Corporation. *Microsoft Virtual PC.* 2007. http://www.microsoft.com/windowsxp/virtualpc/

[24] Microsoft Corporation. *Windows Vista: User Account Control.* 2006.
[25] National Security Administration. *Security-Enhanced Linux.* 2007. http://www.nsa.gov/selinux/.
[26] Donald A. Norman. *The Design of Everyday Things.* Doubleday, 1988.
[27] Novell Corporation. *AppArmor.* http://www.novell.com/linux/security/apparmor/.
[28] OASIS. eXtensible Access Control Markup Language (XACML) version 2.0. 2006.
[29] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proc. USENIX Security Symposium.* 1998.
[30] C. Powers and M. Schunter. Enterprise Privacy Authorization Language (EPAL 1.2). W3C Member Submission, 2003.
[31] Sysinternals. *Rootkit Revealer.* 2006. http://www.sysinternals.com/Utilities/RootkitRevealer.
[32] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The FLASK Security Architecture: System Support for Diverse Security Policies. In *Proc. USENIX Security Symposium.* 1999.
[33] Marc Stiegler, Alan H. Karp, Ka-Ping Yee, and Mark Miller. *Polaris: Virus-safe Computing.* Technical Report. Hewlett-Packard, 2004.
[34] Sun Microsystems. HotJava: The Security Story. 1995.
[35] Sun Microsystems. Java Security Overview. 2007. http://java.sun.com/javase/6/docs/technotes/guides.
[36] Symantec Corporation. *Symantec Norton Antivirus.* 2006. http://www.symantec.com.
[37] The Snort Project. *Snort, The Open Source Network Intrusion Detection System*. 2006. http://www.snort.org/.
[38] The Tripwire Project. *Tripwire host-based IDS. 2007.* http://sourceforge.net/projects/tripwire/.
[39] Michael C. Tschantz and Shriram Krishnamurthi. Towards Reasonability Properties for Access Control Policy Languages. In *Proc. ACM SACMAT.* 2006.
[40] J.D. Tygar and Alma Whitten. Why Johnny Can't Encrypt. In *Proc. USENIX Security Symposium.* 1999.
[41] VMWare Corporation. *VMWare.* 2007. http://www.vmware.com.
[42] VMWare Corporation. *Virtual Machine Communication Interface.* 2007. http://pubs.vmware.com/vmci-sdk/VMCI_intro.html.
[43] David Wagner and Paulo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proc. ACM Conference on Computer and Communications Security.* 2002.
[44] David R. Wooten. Securing the User Input Path On NGSCB Systems. In *Microsoft WinHEC.* 2004. http://download.microsoft.com/download/1/8/f/18f8cee2-0b64-41f2893da6f2295b40c8/TW04055_WINHEC2004.ppt.
[45] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. USENIX Security Symposium.* 2006.
[46] Ka-Ping Yee. Aligning Security and Usability. *IEEE Security and Privacy Magazine*, September 2004: 48 – 55.
[47] Doug Beck, Binh Vo, and Chad Verbowski. Detecting Stealth Software with Strider GhostBuster. In *Proc. Int. Conf. on Dependable Systems and Networks.* 2005.

APPENDIX A. USER INTENT CONCEPTS

This lists the intent concepts inferred by our prototype implementation. While we have implemented all of these concepts in our inference module, we have not yet performed experiments on policies making use of them all, nor have we comprehensively covered all ways that they might be expressed via the Windows user interface.

file_open(*Filename f , Process p*)
The user, interacting with process *p*, intends to open file *f*. This concept is inferred when a user selects a file or set of files in a file open dialog box.

file_save(Filename *f*, Process *p*)
The user, interacting with process *p*, intends to save file *f*. This concept is similar to the file open concept, and is inferred when users indicate their intent to save a file or set of files (through a save dialog). The inference engine distinguishes between open and save dialogs based on the text displayed in their title bars and the API calls used to create them.

file_rename(Filename *f1*, Filename *f2*, Process *p*)
The user, interacting with process *p*, intends to rename file *f1* to file *f2*. This concept is inferred when a user changes the name of the file within a program' user interface (currently, in a file selector dialog).

file_move(Filename *f1*, Filename *f2*, Process *p*)

The user, interacting with process *p*, intended to move file *f1* to file *f2*. This concept is similar to file renaming, and is inferred when a user moves a file within the user interface.

execute(Executable *e*)

The user intended to execute program *e*. This concept is inferred when a user activates a program for execution within the user interface.

program_install(Filename *f*, Executable *e*)

The user intends to install file *f* so that it is associated with program *e*. This concept is inferred when an installation dialog is detected, via name or text (an imperfect solution for the time being). It indicates that a user intends to install a particular set of files and executable programs as a related group. We developed this concept in order to make our candidate policies capable of understanding the implicit relationship between files and programs installed by the same installer program.

APPENDIX B. BENIGN PROGRAMS

This lists the programs used in our contrived benign program testing, as described in section 4.3. Programs were tested using a script that randomly selected elements of their user interface over an 8 minute period in order to exercise the application. In particular, the script was designed to open files if such a menu option was available.

Application Name	Integrity Policy Violations (# in 8 min)
acrord32.exe	0
ad-aware2007.exe	9†
calc.exe	0
csdiff.exe	0
devenv.exe (2003)	2*
devenv.exe (2008)	3*
dkicon.exe	0
eclipse.exe	0
excel.exe	0
firefox.exe	0
gimp-2.4.exe	0
googleearth.exe	3*
gvim.exe	0
hypertrm.exe	0
iexplore.exe	0
itunes.exe	0
msaccess.exe	0
mshearts.exe	0
mspaint.exe	0
mspub.exe	0
notepad.exe	0
outlook.exe	0
picasa2.exe	3*
pictureviewer.exe	0
pinball.exe	0
poweriso.exe	0
powerpnt.exe	0
quicktimeplayer.exe	0
qw.exe	3*
realplay.exe	0
smartftp.exe	3
sol.exe	0
steam.exe	3*
thunderbird.exe	0
tomcat6w.exe	0
truecrypt.exe	0
ttkvwr.exe	0
visio.exe	0
vlc.exe	3*
vmware.exe	0
windbg.exe	0
windlg.exe	2*
winmine.exe	0
winword.exe	3*
wmplayer.exe	3*
wordpad.exe	0

Remarks:
†: Deleted spyware-associated files in violation of policy.
*: Embedded Internet Explorer browser modified IE history/cookies.