HACK: A Health-based Access Control Mechanism for Dynamic Enterprise Environments

Chenjia Wang
Department of Computer Science
Wayne State University
chenjia.wang@wayne.edu

Kevin P. Monaghan
Department of Computer Science
Wayne State University
kpmonaghan@wayne.edu

Weisong Shi
Department of Computer Science
Wayne State University
weisong@wayne.edu

***Abstract*—Current access control schemes focus on the user and their rights and privileges relating to the access to both initiating functionality and accessing information. This approach, while appropriate with respect to access control for the user, misses a very important aspect - the software itself. In this paper, we propose HACK, a health-based, adaptive access control scheme, that provides for both the machine and its software to act on behalf of the users during access. Paramount is that the software itself is included as part of the access control determination. The health of software can be determined when the user attempts to create a new process executing that software. HACK checks its own information about the software to determine its health and can also ask neighboring machines on the network running the same software to provide a health status. Lastly, HACK adapts the access control based on the behavior of the software in response to certain events.**

***Index Terms*—Computer Security, Access Control, Health.**

I. Introduction

With the growth of heterogeneity in the mobile computing environment, secure access is becoming more challenging in design.[5] Laptop, notebook, tablet, and pocket computers, and other mobile computing devices, have been widely used in the enterprise environment but the attention paid to the challenge of securing the computing environment is far from enough. Actually, according to the Redefining Personal Computing with Virtual Computing talk given by Professor Lam in 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 78% use personal computers for work and 43% use work computers for personal use. [3] In the logical perspective, access control decides whether to grant the access right of the object to the principal and in the paper [4], Lampson et al. propose the concepts, protocols and algorithms for access control in distributed systems. The security level of the existing access control mechanisms are either lower than the expectation, which causes the existence of potential risk, or extremely higher than it should be, which has seriously limited the privileges of the user. The conventional role-based access control and rule-based access control mechanisms seem to be unqualified to meet the new requirements posed by the mobile computing environment. Even the extended access control mechanisms of them, such as temporal role-based access control (TRBAC) [1] which supports periodic role enabling and disabling and temporal dependencies among such actions, and generalized role-based access control (GRBAC) may also have some difficulties to face the challenge. Team-based access control (TBAC) is another access control mechanism and the extension access control mechanism of it, the C-TMAC [2] would collect the contextual information including time of access, the location from which access is requested, the location where the object to be accessed resides, transaction-specific values that dictate special access policies, and so on. However, it does not touch on the key component of the contextual information of the client machine which we believe is the health state of the machine. Under this situation, a new adaptive and secure mechanism for access control is highly demanded.

In this paper, we propose HACK, a health-based, adaptive access control scheme. Our approach with HACK is unique in the following ways: we focus on the health of the machines with respect to access control, not the security credentials or privileges of the user; our approach utilizes both a local and community-based check for health, i.e. the status of software is determined not only by the machine running the software but also by its neighboring community; and lastly, our approach is adaptive, i.e. events that take place can alter the state of the machine and can dynamically alter the health of software and therefore its access control. Our key contributions are in the detection of malicious software using a hash of the file contents, the determination of software behavior by a community check, and in the adaptive behavior of the machine based on current state and healthy-based access control events. The remainder of the paper is organized as follows. The design of HACK is presented in Section 2. Section 3 describes the implementation of HACK and the performance evaluation is detailed in Section 4. Sections 5 and 6 cover discussion and related work, respectively. Finally,the conclusion is presented in section 7.

II. Design

The design of HACK provides for a means to ensure the program that is attempting to be executed is valid, that the location from which it is run is appropriate and makes sense, that the program being run is the one that was accepted as being safe, and that the neighboring community agrees with the safety of the program. All these things affect the health of the machine and the machines it is connected to on the network. The design also takes into account the state

of the machine and the different events that can take place and change that state. It must provide a means to identify the programs and their locations which are considered safe and acceptable to execute. And lastly it must provide a way to ask its neighboring community about the reference for a newly created process if the local machine cannot validate the program by itself.

A. Overview of HACK

In order to provide an adaptive capability, the states of the machine and the events that can take place on the machine to change those states must be identified. The states of the machine are simply: *healthy,intermediate healthy and unhealthy*. The healthy state signifies that no adverse events have taken place which would have degraded the state of the machine. The intermediate healthy state signifies that one or more adverse events have taken place which degraded the health and state of the machine. The unhealthy state signifies that one or more adverse events have taken place which further degraded the health of the machine sufficiently where an administrator might be called in, the processes terminated on the machine, or the machine powered down to preclude the negative state from propagating on the network. The no state state signifies a starting point for configuration assuming the administrator does not want to start a machine in a particular state. This is indicative of the machine earning its health rating every time it starts.

The events that can take place on the machine are: *bad path, bad hash, bad process from neighbor, and no event*. The unhealthy process event signifies that an unhealthy process has been identified, independent of whether or not it has been terminated. The bad path event signifies that a process with an appropriate name has been executed from a path not associated to that program. For example, this translates to someone executing calc.exe or notepad.exe on the Microsoft Windows platform but from a directory not associated to the operating system or its programs. The bad hash event signifies that a program was run with an acceptable name from a known location but that the bytes that make up that program do not match up with ones recorded when the program was deemed safe. The bad process from neighbor event signifies that a neighbor within the community deemed a program as unsafe. Lastly, the no event event basically signifies any event for which a state change is not required.

1) Name, Path and Hash based Malicious Software Detection: In order to detect a malicious piece of software, several things must take place. First, the event of when a new process is created must be captured. Second, the name and path of that process must be determined and checked against a master list of allowable processes and their associated locations. Third, the actual executable code from the process is hashed to check against the master list to determine whether or not the binary file has been changed or tampered with since the hash was initially calculated.

In order to identify when a new process is created, a hook or call-back function is called to the operating system registering a function in HACK to call when a new process is created. This allows the capture of the event when the process is actually created, loaded, and executed in memory. It is important to note that HACK places a hook into the operating system in order to identify the event when the process is created, loaded, and executed in memory. This is in contrast to identifying the creation of a new process after it is loaded into memory and executing, which allows for the possibility of the newly created process to cause damage before HACK can be notified and, if appropriate, the software terminated in the case where it is malicious.

Once a new process is created, HACK is notified of its creation via the operating system call-back function and determines its name and path. The path of the executable of the process is important because it identifies from where the process was executed, or said another way, which specific program was executed. If the process name and path match an entry in a list of safe processes, the check continues. If not, the process is deemed malicious and terminated.

Now that the process name and path match an entry in a list of safe processes, HACK reads the executable code into memory and performs a hash check against it. If the hash matches the value in the list of safe processes, the check continues. If not, the process is deemed malicious and terminated since it must have been tampered with considering the hash of its executable code no longer matches a known hash of a safe version of the software.

Once the name, path, and hash of the executable code of the process have all been checked, if the state of the system is healthy, the process is allowed to load and execute unabated. If the state of the system is intermediate , the next step is taken: to check the maliciousness of the process with your neighbors. When the state is less than healthy, its always a good policy to ask the neighbor.

2) Community-based Malicious Services Checking: Depending on the state of the machine, community-based malicious services checking can take place. Community-based checking provides the ability to ask a neighboring machine, henceforth referred to as a neighbor, if they consider the process healthy. If only a single neighbor is asked, then their sole input can provide the determination of the health of the newly created process. If more than one neighbor is asked, a vote can take place to provide the determination of the health of the newly created process.

3) Adaptive Healthy-based Access Control (upgrading and downgrading): An adaptive access control method allows for the ability to change the response to the same security events based on the state of the machine. For example, if event A occurs, e.g. a new process is created, and the state of the machine is state B, e.g. healthy, then HACK will allow the process to execute unabated. However, if the identical event occurs but the state of the machine is state C, e.g. intermediate which is the status after some suspicious process has been found then HACK may initiate a community-based check to determine whether or not to terminate the process.

As events occur, the state of the machine can change.

The simplicity of the design of HACK is that the state machine module only requires that you register which event took place. Recall the allowable events are: bad path, bad hash, bad process from neighbor, and no event. So as events take place, HACK registers that event with the state machine module and it performs all the rule checking itself and, if appropriate, can also initiate state machine change. This allows the actual events that take place as the user initiates requests to execute various processes to adaptively change the state of the machine, i.e. the health of the machine is upgraded (to a better health) or downgraded (to a worse health).

B. Detailed Design of HACK

The three main portions of HACK, malicious software detection, community-based checking, and adaptive behavior are all designed as their own modules. This provides for ease of maintenance, minimizes the propagation of defects, and reduces complexity.

1) Malicious Services Checking: Even computers within the Local Area Network (LAN) may experience different processes and the information of their experience is helpful to detect a new malicious process. It is also easy to exchange information among the computers within LAN. The transmission method used in HACK to exchange information is based on multicast which does not consume too much bandwidth on the network. Assume a new process is created and that a community-based check is initiated. The details of this check will be discussed in next section.

When the community-based check begins, it will send the information on the newly created process, i.e. name, path, and hash value, to its neighbors. Next we introduce three kinds of neighbors: random neighbors, most trustworthy neighbors, and nearest neighbors. Random neighbors are named after the way we choose the neighbors, which, not surprisingly is at random. Most-trustworthy neighbors are those computers which have experienced the initiation of the most new processes and therefore the health reference from them is more reliable. Nearest neighbors are those neighbors that are closest to the machine that requested the community-based check. Once the neighbors receive the information, they will check the information with their own white-list of processes. If the neighbor has seen this process before and it does not have any suspicious behavior, it will send back the reference as being good. If the neighbor has not seen this process before, it will send back the reference as being bad. Depending on the response, the machine which started the community-based check will respond differently. If the reference was good, the process would be allowed to continue running unabated. If the reference was bad, then the process would be terminated immediately.

2) Adaptive Protocol: In order to ensure adaptive behavior with respect to security events, HACK uses a state machine with a straightforward interface where events (bad path, bad hash, bad process from neighbor, and no event) are simply registered when they take place. Given that HACK is notified when a new process is created, HACK registers these events as they take place. The state machine will automatically change state based on the rules identified. For example, if the machine is in the state healthy and the event bad process from neighbor takes place, the state of the machine is automatically changed to intermediate healthy. These rules are easily configured. The state machine changes state based on current state and security event. This state machine can easily be altered to provide different behavior given current state and security event. Figure 1 shows the default state machine as defined by HACK.

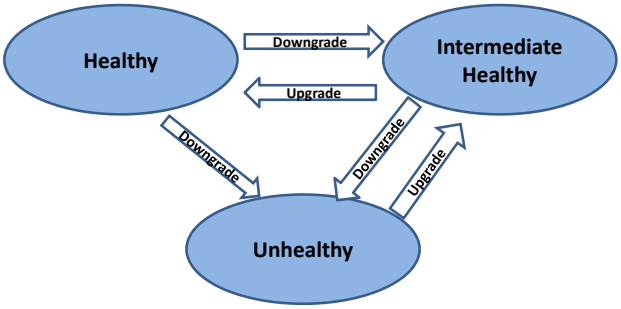

Fig. 1. State Transfer Diagram.

III. Implementation

We implmented HACK on Microsoft Windows XP, service pack 2 and the programming language used for HACK is C++ . The HACK component contains both the client and the server (with respect to community-based checking).

A. Overview

Figure 2 presents the working environment for our HACK project. From the figure we can find that, HACK can handle heterogeneous environment which includes PDA, laptop and desktop. Actually, figure 3 is a typical snapshot of the mobile computing environment. HACK is composed of 3 modules: client sensor module, client access control module and server module. Actually, each machine is playing both the role of server and client. The relations and the function included in each module are presented in figure 3.

B. Client Sensor Module

The client sensor module basically has 2 functions including hash/path based malicious software detection and community-based malicious services checking function. Hash/path based malicious software detection is designed and implemented as follows: Both hash and path checks are based on the white-list we created for them. In the white-list, we would have the information for allowed process. The information is composed of the process name, default path and the hash value based on its content. The path check is aimed to judge whether a newly created process has the exact path as it displayed on the white-list. As for the hash check function, SHA-1 (Secure Hash Algorithm) which is safer than MD5 and produces a message digest that is 160 bits long is used to calculate the hash value for the newly created process. Based on the hash value stored

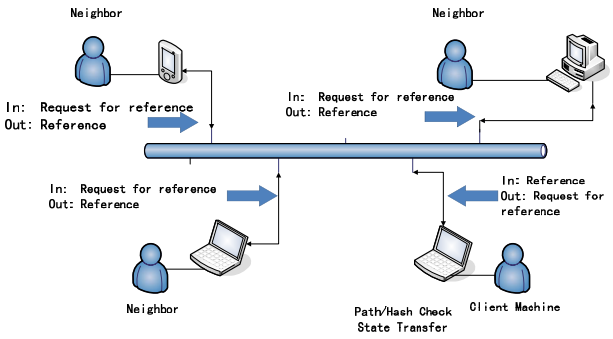

Fig. 2. Overview of Hack Design.

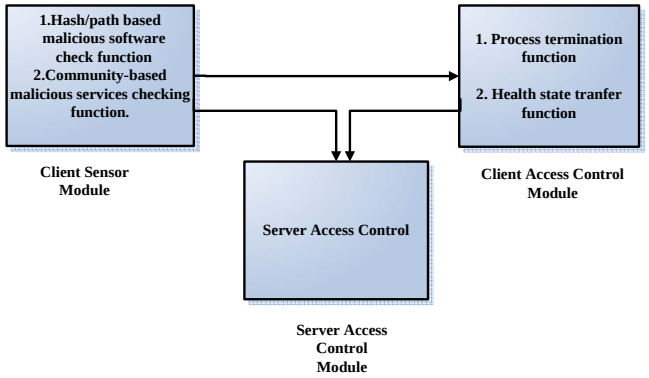

Fig. 3. 3 modules of HACK.

in the white-list, it is easy to tell whether the newly created process is the same one which is allowed to be running on the machine. Hash/path based malicious software detection function is within the client machine in which the process has started. In another word, it is a local-check function.

Different from the hash/path based malicious software detection, the community-based malicious service checking function is a network-based check function. Figure 4 is the diagram for the community-based malicious service checking function.

From figure 4, it is obvious that each users machine would either request for others reference or give reference. As we mentioned before, the communication between them is implemented by multicast. The request would be sent to all the neighbors within the subnet and all the neighbors would return their reference to the machine which started the multicast. However, not all the references would be considered for the starting machine to make the final decision on the judgment of the newly created process. Only the references from the most-trustworthy neighbors would be taken. So actually not only the references was sent back but also the metadata of the neighbors would be returned. In detail, the metadata includes

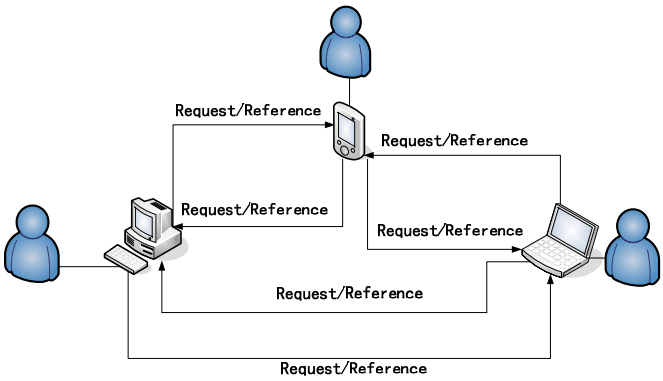

Fig. 4. Diagram for the community-based malicious service checking function.

the IP address of the neighbors as well as the number of the process it has seen. As long as the starting machine receives all the reference, it would pick up the references from the most-trustworthy neighbors and decide the next action it should take from which the client access control module starts.

C. Client Access Control Module

The two functions included in the client access control module are: 1.)Termination of process function 2.)Health state transfer function. Once the client machine finished the hash/path based malicious software detection and community-based malicious services check, it will carry out the termination of process function if necessary or the health state transfer function.

For the termination of process function, it will kill the process if it meets the following requirements: 1.) the process has the same name as one of the allowed process but has a different path. It is quite common that the malicious process would try to pretend to be a normal common process while its path is totally different from the normal one. 2.) the process has the same name and path as the normal process but its hash value is different from the one on the whitelist. Even the process has the same name and path as the normal process, sometimes the digest of it may have a difference from the one on the whitelist. Thus, the hash check demonstrates its importance in helping us to identify the malicious software or service. Using the SHA1 algorithm, the client access control module would generate a 160 digit value for each process and based on the comparison with the one on the white-list, it would let the administrators know whether the content has been modified.3.) the last scenario which would leads to the termination function is that the name of the process is not on the white-list and the reference from the trustworthy neighbors are bad. It is obvious that in this scenario there are 2 requirements before the module finally terminates the process and the second one is not based on the judgment from the local machine.

As for the health state transfer function, we have defined 3 states for each machine which are healthy, intermediate healthy and unhealthy. Based on each name, it is easy to guess that

the machine in the healthy state would have most priorities in the access and unhealthy state would have very limited or no access. The intermediate healthy state is the state we would put emphasize on. Differ from most access control mechanism which either allow or do not allow access, the intermediate state enable the administrator to observe a certain process for a while before taking any further action. Actually, the transfer or upgrade/ downgrade can also be automatically implemented by HACK itself. As long as it meets the policy we have set for the HACK, the state of the client machine would be transferred or maintained by the client access control module.

IV. PERFORMANCE EVALUATION

In this section, we present the experimental evaluation of our HACK system. The evaluation experiment is composed of two parts. First, the validation test of the function of HACK. Second, the overhead of HACK. In the validation test, we will use several cases to test the Path/Hash malicious software detection, Community-based malicious services checking and Adaptive health-based access control function as we designed. As for the overhead, we will test the delay of the execution of HACK for Path/Hash Malicious software detection and Community-based malicious services checking. The experiments were running on the environment as follows. Four desktops with Windows Professional Service Pack 2 and the hardware configuration is Intel Pentium4 2.6 GHz, 512 MB memory, 80G harddisk.

A. Validation Test

The validation test is designed to test all three function we proposed. We started the test with the validation of Path/Hash Malicious software detection. In this test, we used Microsoft Paint program as the 1st case. It is on our white-list with all the information of it including name of this process, path and hash value. We started our HACK program first. After its initialization, we are ready to start the test. We try to execute the MS paint and our client sensor module intercepted the execution of MS paint. It searched the white-list for MS paint and since it is on the white-list, the program displayed it is on the white-list. Then it compared the path and hash value of the process and they are correct. So it will let MS paint run without any further action and the state is still healthy.

The 2nd case is to use the Calculator program which is in the Accessories folder of Windows as the testing object. We created a shortcut of it and linked it to another program so once it was executed , it lead to another program. By doing so, it would have a different hash value. In the test, HACK correctly found out the different hash value and terminated the application. Also, the state of the machine was transferred from Healthy to Intermediate Healthy which means Adaptive health-based access control also worked properly.

Then we tested the Community-based malicious services checking. We randomly picked up a machine among those 4 machines and removed the information of Wordpad application from its white-list. So once we tried to execute the Wordpad on it, the HACK intercepted and it could not find it on its local white-list, therefore, the Community-based malicious services checking started. It sent out the request to verify the Wordpad application and its neighbors received the request. They sent back the reference and since it was on the white-list of other machines, the reference was Good. So the machine which sent out the request allowed Wordpad to continue running and no transfer occured.

	1	2	3	Average
Valid Path and Hash	15.0 ms	15.0 ms	16.0 ms	15.3 ms
Invalid Path/Hash	344.0 ms	297.0 ms	187.0 ms	276.0 ms

Fig. 5. Overhead of Path/Hash Malicious Software Detection.

The final case was used to verify whether HACK could take corresponding action once it received the reference as Bad from its neighborsWe used EdittPlus Text Editor as the object. The information of it was not on any machine and none of them has run this one before. So it should not be found on the white-list and all the reference from the neighbors should be 'bad'. In the test, after we try to execute the EditPlus program, the HACK intercepted it. It could not find it on the white-list so it sent out the request. The neighbors received it and they could not verify it either so they send back Bad. HACK then terminated the application and transferred the state from Healthy to Interme

B. Performance Overhead

The overhead of Path/Hash Malicious software detection is calculated from the beginning of the attempted execution of the application to the end of the corresponding action has been taken. From the table we can see that HACK would not cost an obvious overhead and the reason why the overhead is much bigger once the application has an invalid path/hash is that HACK would invoke another program to kill the process which could take sometime.

The overhead of Community-based malicious services checking includes the time once the request was sent out until the time the reference received by the machine which sent out the request. It is clear in the table that there is no big difference whether we test on three or four machines and both of them were small. So we believe we could deploy HACK on more machines with almost no increase in overhead and a more trustable reference due to the increase in the number of neighbors.

	1	2	3	Average
3 machines	16.0 ms	31.0 ms	15.0ms	20.7 ms
4 machines	16.0 ms	16.0 ms	31.0 ms	21.0 ms

Fig. 6. Overhead of Community-based Malicious Services Checking.

V. DISCUSSION

The current HACK approach may still be at risk if the malicious program can develop corresponding evasion technique and inevitably, the design may still have some deficiency. In this section, we are going to discuss about the potential risk and the countermeasures.

A. How to secure the HACK while it is running on the top of the client machines

So far, HACK is running on the client machines as an application and thus if the malicious programmers are aware of the existence of HACK, they may attempt to modify the key component or even subvert the whole system. Currently, the most likely method they would use includes modifying the white-list and falsifying the reference from the neighbors in community based check.

1) Modifying the white-list: Since we would push the white-list to each client machine and let them do the comparison by reading the white-list stored locally, one possible approach to distort the judgment of HACK is to modify the content of the white-list. The malicious program could locate the path and folder of the white-list and modify the information which is related to its own. If they could successfully update the white-list based on their demand then the HACK would make a wrong judgment and instead of terminating the malicious application, it would allow the current malicious application to be run on the local client machine.

The solution to this kind of distortion could either be implemented by generating a hash value for the white-list itself, storing it in the server to eliminate the possibility for the malicious program to get access to it and whenever a white-list based comparison is executed, the hash value of the white-list would be checked. Thus, we could ensure that the white-list is always trustable. It would be easy to implement this approach but it would be at the cost of the increased delay.

2) Falsifying the reference from the neighbors in community based check: Another possible approach to distort the judgment of HACK is to intercept and falsify the reference from the neighbors. Besides the local hash/path checking, community-based check is also utilized here to help HACK to detect the malicious application. Thus, another method to affect the validity of HACK is to falsify the reference. Although it would be hard to modify the reference from the neighbors without access to the source code, it is possible that the malicious program could try to intercept the reference sent back from the neighbors and modify it from Bad to Good by which it would make itself appear to be a normal process already run on a certain machine.

However, it would be difficult for the malicious program to intercept the message since it could only intercept the reference either from the most-trustworthy neighbor when it created the reference or falsify it when the reference reaches the machine which starts the community-based check. To intercept it in the most-trustworthy neighbor, the malicious application needs to be able to know where the neighbor is and locates the buffer which we are using. This would be extremely hard to implement. Although it would be relatively easier to modify the reference directly in the client machine which starts the community based check, it still would be difficult for it to implement this.

3) False Positive and False Negative: Although HACK performed well among all the scenarios we have designed in the evaluation part, there is still possibility that HACK would make false positive or false negative. For the false positive, we are concerned about the update for each application which could cause the change of the content and trigger the false positive of HACK. Another possible inducement to false positive would be moving a certain application to a different location which would cause the HACK to falsely terminate the application due to the policy we have set. The solution to this problem would be the synchronization of the update of white-list. Once an update has been released or the application has been relocated in another folder, the white-list needed to be updated with corresponding information. Although we have not implemented this in the HACK, it would be our future work to include this as part of the whole system.

For the false negative, the possible situations would be those we have discussed in the former section which includes Modifying the white-list and Falsifying the reference from the neighbors in community based check by the malicious application. So we are not going to repeat it here.

VI. RELATED WORK

The two popular access control mechanism in the enterprises environment are centralized access control mechanism and decentralized access control mechanism. It seems that there is no right answer which one is better and pretty much work has been done for these two mechanisms. HACK is more decentralized access control mechanism since each machine itself has the Client Sensor Module and Client Access Control Module which would limit their access to certain resource.

Also, access control constrains what a user can do directly, as well as what programs executing on behalf of the users are allowed to do. In this way, access control seeks to prevent activity that could lead to breach of security.Thus, several kinds of access control mechanisms have been proposed from this perspective such as Role-based Access Control (RBAC)

introduced by Sandhu etc [6]in Rolebased access control models, Distributed Role-Based Access Control (dRBAC) and Temproal-RBAC(extension of RBAC). Team-based Access Control (TMAC) and its extension C-TMAC which extends TMAC by using general contextual information are another two examples[7]. Our HACK system is different from the previous work on the concept of health-based access control. We introduce the 3 states (Healthy, Intermediate Healthy and Unhealthy States) of the machines to grant different access privilege. It is adaptive and more flexible. Especially for the intermediate healthy state, we allow the administrator to monitor the machine in the intermediate state for a while before taking any action which might be improper. Also, different from the most access control mechanism, the Community-based malicious services checking has been proposed and implemented here which enables the communication between certain machines to provide a more reliable reference for the detection of access control instead of just following the static policy or white/blacklist pushed to each machine. Thus, we believe HACK is more reliable for dynamic enterprise environments.

Existing access control schemes are specific to authentication and authorization of the user and the privileges, and as such, are insufficient to provide for and ensure the security of the machine itself. Malicious software can exist unnoticed and reek havoc on systems. We propose HACK, a health-based access control scheme, that fills the gap between existing access control schemes and the need to ensure the health of the machine. HACK provides for the identification of a new process, ensuring that new process is on a white-list, ensuring that the content of the process has not been compromised. As well, HACK provides for a community-based check to ask neighboring machines their input on the health of a new process. Finally, HACK provides for adaptive behavior by allowing different events to change the state of the machine and ultimately its behavior specific to allowing a new process to execute or terminating it.

VII. Conclusion

Existing access control schemes are specific to authentication and authorization of the user and the privileges, and as such, are insufficient to provide for and ensure the security of the machine itself. Malicious software can exist unnoticed and reek havoc on systems. We propose HACK, a health-based access control scheme, that fills the gap between existing access control schemes and the need to ensure the health of the machine. HACK provides for the identification of a new process, ensuring that new process is on a white-list and that the content of the process has not been compromised. As well, HACK provides for a community-based check to ask neighboring machines their input on the health of a new process. Finally, HACK provides for adaptive behavior by allowing different events to change the state of the machine and ultimately its behavior specific to allowing a new process to execute or terminating it.

Acknowledgment

The authors would like to thank Nardina Mein, Tung Nguyen, Shinan Wang, Rod Fiori, James Wurm and Mike Dobson for their technical help in this paper. Chenjia Wang was supported by the generous support from the School of Library and Information Science.

References

[1] E. Bertino and P. Bonatti. Trbac: A temporal role-based access control model. In *ACM Transactions on Information and System Security*, page 191C223, August 2001.

[2] C. Georgiadis, I. Mavridis, G. Pangalos, and R. Thomas. Flexible team-based access control using contexts. In *ACM Symposium on Access Control Models and Technologies*, Chantilly, USA, May 2001.

[3] M. Lam. Redefining personal computing with virtual computing. In *2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Mar. 2009.

[4] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[5] H. Lufei, W. Shi, and V. Chaudhary. Adaptive secure access to remote services. In *SCC08*, 2008.

[6] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. In *IEEE Computer*, page 38C47, February 1996.

[7] P. Sudame and B. Badrinath. Transformer Tunnels: A Framework for Providing Route-Specific Adaptations. In *Proc. of the USENIX Technical Conf.*, New Orleans, Louisiana, June 1998.