Shadow IT Software Development - End User Linux PC Side Mitigation Controls with Mandatory Access Control and Logging

Joshua Nathaniel Haupt - joshua@hauptj.com

Disclaimer

The research, observations, and proposals presented in this paper are my own and do not represent any company or organization I have worked for or with.

Abstract

Shadow Information Technology (IT) encompasses all hardware, software, or other solutions used by employees within the organizational ecosystem that have not received formal approval from the IT department. Shadow IT is a problem in many large organizations that handle sensitive information and infrastructure. [1] Shadow software development is a subset of shadow IT, where undocumented applications are developed and deployed within enterprise environments. This paper outlines the use of Mandatory Access Control and proper logging of attempted usage of development tools, such as controls that can be implemented on the end-user or client side, to mitigate the risk of shadow application development.

Risks

The risks associated with shadow IT are quite costly, with the global average cost of a data breach in 2024 being $4.88 million. The costs include expensive compliance violations, as well as legal and reputational risks, especially if sensitive data is compromised. [2]

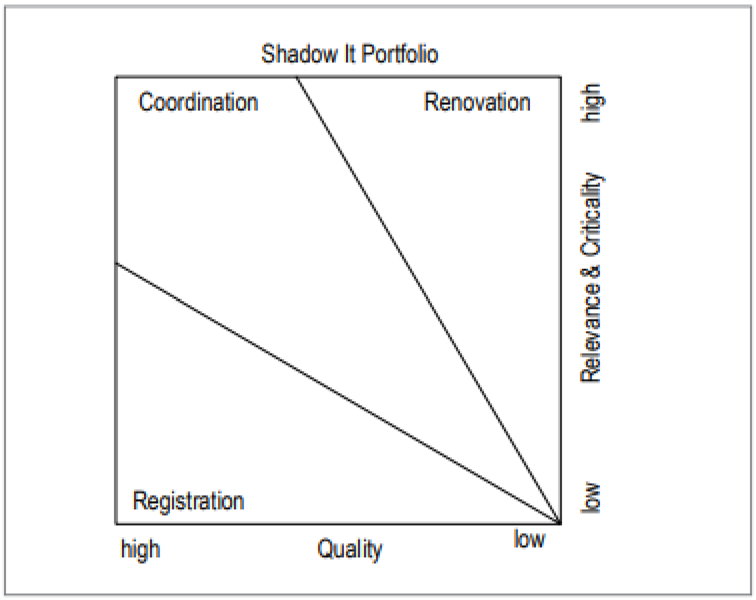
Both data breaches and integrity failures typically stem from mishandling data at rest, in transit, or in use.

Data breaches can result from improper data handling, such as using sensitive customer information in non-secure testing databases, utilizing unauthorized external databases, employing vulnerable third-party software libraries, and hardcoding sensitive credentials instead of properly vaulting them using a platform like Hashicorp Vault.

Data integrity compromise is a significant risk associated with shadow IT, as such solutions often lack robust testing and safeguards, and are frequently of lower quality. For instance, in a distributed Robotics Process Automation (RPA) project, a shared Microsoft Access database held critical workflow data. An accidental mouse movement deleted essential rows, disrupting the entire process mid-execution.

Inadequate controls against shadow software development can provide new threat vectors for ransomware and insider attacks, as unauthorized tools may be leveraged by malicious actors.

Shadow IT Implementation - Quality vs Risk Analysis [3]



From this diagram, by Zimmermann [3], you can see that how a shadow IT implementation should be handled is based on the correlation between the quality of the implementation and the relevance and criticality of the implementation. If a shadow IT implementation is of low quality, but high relevance and criticality, mitigation efforts often require renovation, or fixing, and if necessary, rebuilding it. On the other hand, if the quality of a shadow IT implementation is very high, it may be safe enough to just register the implementation. Instances of shadow IT implementations that are in between require some coordination, as some renovation or fixes may still be necessary before they can be registered for proper utilization.

Motives - IT Side

On the IT side, shadow software development often results from developer frustration with the available tools and resources within the enterprise. This can be due to a lack of adequate testing environments or development tools, such as Integrated Development Environments (IDEs) or debugging tools, that are available for use. The frustration with, or even lack of, internal developer platforms is another cause. Obviously, if no internal development platform exists, it can be a chaotic free-for-all. However, even if one does exist, it is often the case that developers may not understand how to properly utilize it, often due to inadequate training. Additionally, they may also be frustrated with a lack of support for using the platform.

Motives - Business Side

On the business side, there may be a lack of support from IT or other development teams to implement necessary solutions for the business. As a result, business users may experiment with no-code / low-code platforms as well as Robotic Process Automation (RPA).

Control Objectives

The proposed controls to be discussed for preventing shadow software development must satisfy the following conditions. 1.) They must ensure that only authorized users can access development tools. 2.) Both unauthorized and improper usage of development tools needs to be logged to enable upstream auditing. These controls can be implemented on end clients and on the network.

Controls - End User Personal Computers (PCs)

1. Require all network traffic to flow through the enterprise network to prevent the use of unapproved web and cloud services. If an end-user device is used remotely, all of its network traffic should be routed through an enterprise VPN. This control and its implementation are a prerequisite for
2. Only allow approved software to be requested, downloaded, and installed from trusted internal repositories.
3. Restrict user access to development tools.
    a. Restrict the ability to request development tools to authorized users only.
    b. Restrict the development tools users can request to only those that are necessary for their job function.
    c. Through Mandatory Access Control [a], restrict the ability to execute or run development tools based on the user's group memberships. Unauthorized attempts to use development tools should be logged for upstream auditing. On Linux and other Unix-based systems, this can be implemented with proper permission configurations that are owned by the root user, allowing them to be centrally managed. These permissions need to restrict execution permissions to development tools to users of a specifically designated developers group. Further enforcement of proper usage of development tools and logging attempts at improper usage can be handled on Linux systems with centrally managed kernel security modules, such as AppArmor. Security-Enhanced Linux (SELinux) was evaluated in my experiments; however, during my lab experiments, I did not find it to be a practical tool for restricting the use of development tools on development systems used by individual users.

Lab Setup

For the lab environment in which I tested my hypotheses, I used two separate laptops: one running Fedora 41 Workstation Edition and another running Ubuntu 24.04.3 LTS.

End User PC Side Control Implementation - Use Mac to Restrict Access to Development Tools

The utilization of MAC to restrict development tools, such as IDEs, consoles, and text editors to write to and execute code in designated development directories is an additional control that should be implemented as it relates to PO.5 - 'Implement and Maintain Secure Environments for Software Development of the SSDF. [6] This is more AppArmor, as both allow for fine-grained policies that can restrict who can access an application and where applications can write to. A drawback is having to maintain security policies for every development tool. Another drawback is that implementation can be challenging for specific groups that have access to it. However, this drawback can be mitigated by assigning developers a secondary account or secondary developer machines. Both of these controls not only relate to PO.5, but also enable streamlined auditability, as all activity involving these secondary accounts or machines can be logged and monitored as development activity.

In my lab environment, on both the Fedora and Ubuntu machines, I was able to successfully restrict access to development tools, such as the nano, emacs, editors and the clang, gcc, and g++ compilers by setting the group ownership of binaries of the relevant binaries in /usr/bin/ to the "devs" group, and then removing the execute permissions for non owner or group members. I was also able to implement logging by configuring audit.d to monitor for failed execution, or "execve" system calls for non-system and non-root users.

Here are the commands to restrict development application binaries to members of a specific group, in this case, the "devs" group.

```
sudo groupadd devs
sudo usermod -aG devs josh
sudo chown root:devs /usr/bin/nano
sudo chmod 710 /usr/bin/nano
sudo chown root:devs /usr/bin/emacs
sudo chmod 710 /usr/bin/emacs
sudo chown root:devs /usr/bin/clang
sudo chmod 710 /usr/bin/clang
sudo chown root:devs /usr/bin/gcc
sudo chmod 710 /usr/bin/gcc
```

Here are the required rules that need to be added as a rules file under /etc/audit/rules.d/.

```
sudo cat /etc/audit/rules.d/denied.rules

-a always,exit -F arch=b32 -S open,creat,truncate,ftruncate,openat,execve -F
exit=-EACCES -F auid>=1000 -F auid!=-1 -F key=access

-a always,exit -F arch=b32 -S open,creat,truncate,ftruncate,openat,execve -F
exit=-EPERM -F auid>=1000 -F auid!=-1 -F key=access

-a always,exit -F arch=b64 -S open,truncate,ftruncate,creat,openat,execve -F
exit=-EPERM -F auid>=1000 -F auid!=-1 -F key=access

-a always,exit -F arch=b64 -S open,truncate,ftruncate,creat,openat,execve -F
exit=-EACCES -F auid>=1000 -F auid!=-1 -F key=access
```

This is what a user outside of the "devs" group will see when they try to run a restricted development tool, which in this case is nano.

```
nondev@josh-ubuntu-laptop:~$ nano

bash: /usr/bin/nano: Permission denied
```

This is an audit log that is generated when a user named "nondev" who is outside of the "devs" group tries to run a restricted development tool such as nano.

```
sudo cat /var/log/audit/audit.log

type=SYSCALL msg=audit(1759799497.953:1669): arch=c000003e syscall=59
success=no exit=-13 a0=5a8a296a7c00 a1=5a8a296a7ba0 a2=5a8a2968d190
a3=5a8a296a7ba0 items=1 ppid=15098 pid=18160 auid=1000 uid=1001
gid=1001 euid=1001 suid=1001 fsuid=1001 egid=1001 sgid=1001 fsgid=1001
tty=pts4 ses=3 comm="bash" exe="/usr/bin/bash" subj=unconfined
key="access"ARCH=x86_64 SYSCALL=execve AUID="josh" UID="nondev"
GID="nondev" EUID="nondev" SUID="nondev" FSUID="nondev" EGID="nondev"
SGID="nondev" FSGID="nondev"

type=CWD msg=audit(1759799497.953:1669): cwd="/home/nondev"

type=PATH msg=audit(1759799497.953:1669): item=0 name="/usr/bin/nano"
inode=394303 dev=08:02 mode=0100710 ouid=0 ogid=1002 rdev=00:00
nametype=NORMAL cap_fp=0 cap_fi=0 cap_fe=0 cap_fver=0
cap_frootid=0OUID="root" OGID="devs"

type=PROCTITLE msg=audit(1759799497.953:1669): proctitle="bash"
```

In my lab environment, I successfully tested this using AppArmor on the Ubuntu machine. I configured two separate users, one outside of a devs group and another in the devs group. The user outside of the devs group was restricted from running most development tools, such as editors and compilers. While the user in the devs group was able to use such tools only while working inside the designated dev directory.

Here is an example of an AppArmor profile configuration that restricts the nano text editor's write permissions to a "dev" directory that is in their home directory by specifically whitelisting the"home/*/dev/" directory and all of its sub directories for the file "owner" which in this case will be the user "josh".owner /home/*/dev/ rw," and " owner /home/*/dev/** rw,"

```
sudo cat /etc/apparmor.d/usr.bin.nano
abi <abi/3.0>,


include <tunables/global>


/usr/bin/nano {
include <abstractions/base>
include <abstractions/bash>
include <abstractions/evince>


/etc/nanorc r,
/etc/nsswitch.conf r,
/etc/passwd r,
/usr/bin/nano mr,
owner /home/*/dev/ rw,
owner /home/*/dev/** rw,
}
```

Here is an example of an AppArmor profile configuration that restricts the clang compiler's write permissions to to temporary directories with "include <abstractions/user-tmp>" as well as the "dev" directory that is in their home directory by specifically whitelisting the"home/*/dev/" directory and all of its sub directories for the file "owner" which in this case will be the user "josh".owner /home/*/dev/ rw," and " owner /home/*/dev/** rw,"

```
abi <abi/3.0>,
include <tunables/global>

/usr/lib/llvm-18/bin/clang {
include <abstractions/base>
include <abstractions/bash>
include <abstractions/consoles>
include <abstractions/opencl-pocl>
include <abstractions/user-tmp>

/etc/ld.so.cache r,
/etc/locale.alias r,
/etc/lsb-release r,
/usr/bin/x86_64-linux-gnu-ld.bfd mrix,
/usr/include/** r,
/usr/lib/llvm-18/bin/clang mr,
/usr/lib/llvm-18/bin/clang mrix,
/usr/local/include/ r,
owner /home/*/dev/ rw,
owner /home/*/dev/** rw,
}
```

This is an example of an audit log output of when a user named "josh", who is in the "devs" group, tries to use a development tool, in this case, the nano text editor, outside of an authorized development directory.

```
sudo cat /var/log/audit/audit.log

type=SYSCALL msg=audit(1759800630.117:1734): arch=c000003e syscall=257
success=no exit=-13 a0=ffffff9c a1=5fc9c5111130 a2=c1 a3=1b6 items=1
ppid=5018 pid=19069 auid=1000 uid=1000 gid=1000 euid=1000 suid=1000
fsuid=1000 egid=1000 sgid=1000 fsgid=1000 tty=pts3 ses=3 comm="nano"
exe="/usr/bin/nano" subj=/usr/bin/nano key="access"ARCH=x86_64
SYSCALL=openat AUID="josh" UID="josh" GID="josh" EUID="josh" SUID="josh"
FSUID="josh" EGID="josh" SGID="josh" FSGID="josh"

type=CWD msg=audit(1759800630.117:1734): cwd="/home/josh"
```

```
type=PATH msg=audit(1759800630.117:1734): item=0 name="./"
inode=15073282 dev=08:02 mode=040750 ouid=1000 ogid=1000 rdev=00:00
nametype=PARENT cap_fp=0 cap_fi=0 cap_fe=0 cap_fver=0
cap_frootid=0OUID="josh" OGID="josh"

type=PROCTITLE msg=audit(1759800630.117:1734):
proctitle=6E616E6F00746573742E747874
```

I was unsuccessful in using SELinux on Fedora to achieve the same results as SELinux's label-based implementation, as it proved extremely difficult to enable many applications, in this case, development controls, to have shared access to a single specific directory.

One limitation is the inability to limit the use of common interpreters, such as Python, which are used for system-related functions. Restricting Python is advised against, as it may cause dependent software to break.

```
sudo aa-genprof /usr/bin/python3

/usr/bin/python3.12 is currently marked as a program that should not have its own

profile. Usually, programs are marked this way if creating a profile for

them is likely to break the rest of the system.If you know what you're

doing and are certain you want to create a profile for this program, edit

the corresponding entry in the [qualifiers] section in /etc/apparmor/logprof.conf.
```

End User PC Side Control Implementation - Secondary "Developer" User Accounts

With this control, users who are members of development groups are assigned secondary accounts from which they can access with '*runas*' on Windows OSes or '*su*' on Unix/Linux OSes. A login prompt to access the secondary profile should be required as an added security measure. These secondary accounts will have the necessary permissions to read from, write to, and execute from designated development directories.

In my lab environment, I tested this on Fedora and Ubuntu. I was able to initially log in a user as a non-developer user outside of the "devs" group; however, I was unable to prevent users from logging directly into a developer who is a member of the "devs" group. Additionally, I was unable to adequately restrict the developer user to only writing to specific development directories using developer tools, as I was unable to granularly restrict write permissions for individual applications.

Due to AppLocker's limitation, in which it is not able to restrict where a whitelisted application is able to write to, as it only allows or blocks applications from running and does not control the behavior of applications after they're launched, [7] I decided to no longer further pursue it as a viable candidate for evaluation.

End User PC Side Control Implementation - Secondary "Developer" Machines

With this control, users who are members of development groups are assigned secondary development machines instead of secondary accounts. These secondary machines can be physical or virtual. With virtual machines, this approach may also allow developers to have limited elevated permissions from within the VM, as the VM will enable more isolation and sandboxing. One major disadvantage is that this approach can be more expensive to implement and manage.

Since I was unable to find a viable solution to restrict where development applications on Windows operating systems can write, this further supports the case for secondary Linux developer machines with AppArmor configured. This is especially the case when a primary Windows machine is required for non-development office work while the software being developed is intended to be run on a Linux or Unix server.

Conclusion

Shadow software development, a subset of shadow IT, can be mitigated through the implementation of adequate end-user controls on personal computers. These controls must ensure that only authorized users can access development tools and use them in a compliant and easy-to-audit manner. These control objectives can be achieved through proper network controls, as well as Mandatory Access Control (MAC), which utilizes centrally managed group permissions to restrict access to developer tools to members of a designated developer group. Additionally, AppArmor can be used to ensure that developer tools are only able to write to designated directories. These controls and their implementations encompass PO.5 - 'Implement and Maintain Secure Environments for Software Development of the SSDF. [6]

Future Work

Further evaluation on SELinux should be considered, as it has not been entirely ruled out as a viable candidate, even though its configuration would likely not be trivial to implement or even maintain for this use case. User experience testing should also be performed using a development environment with these controls implemented, ideally in an enterprise environment.

Definitions

[a] Mandatory Access Control - An access control policy that is uniformly enforced across all subjects and objects within a system. A subject that has been granted access to information is constrained from: passing the information to unauthorized subjects or objects; granting its privileges to other subjects; changing one or more security attributes on subjects, objects, the system, or system components; choosing the security attributes to be associated with newly created or modified objects; or changing the rules for governing access control. Organization-defined subjects may explicitly be granted organization-defined privileges (i.e., they are trusted subjects) such that they

are not limited by some or all of the above constraints. Mandatory access control is considered a type of nondiscretionary access control. [4] and [5]

References

[1] Silic, Mario & Back, Andrea. (2014). Shadow IT â€“ A view from behind the curtain. Computers & Security. 45. 10.1016/j.cose.2014.06.007.

[2] Pimentel, Brandon. â€œThe Cost of Data Breaches.â€ *Thomson Reuters Law Blog*, 11 Dec. 2024, legal.thomsonreuters.com/blog/the-cost-of-data-breaches/.

[3] Zimmermann, S., Rentrop, C., Felden, C. Managing Shadow IT Instances - A Method to Control Autonomous IT Solutions in the Business Departments. Americas Conference of Information Systems, 2014, 1-12.

[4] NIST (2020). *Security and Privacy Controls for Information Systems and Organizations*. https://doi.org/10.6028/nist.sp.800-53r5

[5] mandatory access control (MAC) - Glossary | CSRC https://csrc.nist.gov/glossary/term/mandatory_access_control

[6] Souppaya, M., Scarfone, K., & Dodson, D. (2022). *Secure Software Development Framework (SSDF) Version 1.1â€¯:* https://doi.org/10.6028/nist.sp.800-218

[7] jsuther1974. (2024, January 10). *Security considerations for AppLocker*. Microsoft Learn. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/applocker/security-considerations-for-applocker