Chrome app-bound encryption Service

(formerly: Chrome Elevated Data Service)

A pathway towards a more secure storage mechanism on Windows.

Author: <wfh@, forshaw@> Last Modified: <2021-01-25>
Status: [Draft | Under Review | Reviewed | **Implemented** | Obsolete]

Review

Reviewer	**Status:** 🕑Requested, 🙏 In Review, ✓ Approved
ldap@	🕑Requested

You can also use Google Doc Add-ons such as go/docwrap to track the review state.

One-page overview

On Windows, sensitive data can be stored bound to the logged in user using the DPAPI (Data Protection API). This means that data can only be retrieved by applications running as the same user that originally stored the data. This protects against certain types of attacks, primarily offline attacks (e.g. stolen laptop), or multiple users on the same workstation accessing each other's data.

Sensitive data includes but not limited to, cookies, passwords, and credit cards.

Because any application running as the same user as Chrome can freely decrypt data encrypted with the DPAPI, there have been a number of cases where an application running at the same privilege level as Chrome can read this data. Sometimes this is with the user's tacit permission (e.g. a replacement browser can helpfully import bookmarks, history, cookies and passwords) and sometimes this is without the user knowing (e.g. malware downloaded to the machine can trivially read this data).

The aim of this project is to try and come up with a way whereby further protections could be put in place to ensure that only code running inside Chrome, or applications running elevated can perform these DPAPI operations.

IMPORTANT The aim is not to prevent an attacker who is running at a higher privilege than Chrome from performing these operations - e.g. Administrator, SYSTEM, an enterprise admin, a rootkit or a kernel driver. This is just to prevent an attacker with the same privilege as Chrome from trivially calling the APIs.

Summary

The existing Elevation Service will have new IPCs added to support DPAPI-style operations that are app-bound.
OSCrypt will be augmented to use these new IPCs instead of DPAPI.

Platforms

	Select applicable platforms in **Bold**	Reasoning, if applicable
Desktop	Linux, Mac, **Windows**, Chrome OS, Lacros (go/lacros)	*APIs being discussed are Windows only*
Mobile	Android: Chrome, WebView, WebLayer	
	iOS	

If your design includes/supports Chrome UI change, make sure to read the Design section.

Team

wfh@google.com, forshaw@google.com

PRD

No PRD this is a technical change.

Launch bug

Pending. (When you have a launch bug, stick it here. Leave as “pending” until you do.)

Code affected

components/os_crypt, process startup, installer +tbd.

Design

Background

On Windows, sensitive data can be stored bound to the logged in user using the DPAPI (Data Protection API). This means that data can only be decrypted by applications running as the same user that originally encrypted the data. This protects against certain types of attacks, primarily offline attacks (e.g. stolen laptop), or multiple users on the same workstation accessing each other's data. Chrome already uses the DPAPI in this way, via an interface exposed in components/os_crypt.

"Sensitive data" includes but is not limited to, cookies, passwords, and credit cards.

Because any application running as the same user as Chrome can freely decrypt data stored with the DPAPI, there have been a number of cases where an application running at the same privilege level as Chrome can access this data. Sometimes this is with the user's tacit permission (e.g. a replacement browser can helpfully import bookmarks, history, cookies and passwords) and sometimes this is without the user knowing (e.g. malware downloaded to the machine can read this data).

Non Goals

The aim is not to protect against an attacker who has the ability to run code at a privilege level higher than Chrome e.g. Administrator, SYSTEM, an enterprise administrator, a kernel driver, or an exploit that is able to elevate to SYSTEM.

The current design does not protect against code injection into Chrome.exe to hijack the encryption key or make a call to the Chrome Elevated Data Service.

Proposal (Elevation Service)

Note: See Appendix A for the previous proposal which used the UIAccess executable bit to achieve similar goals.

Chrome already ships the Chrome Elevation Service (located in src/chrome/elevation_service) this is installed when Chrome is installed as system-wide and provides a COM interface to support elevated operations. Currently the only elevated operation that the service exposes is to take a signed CRX and re-install the Omaha update component. This is used for recovery of the updater.

Proposal is to add two new IPCs to this COM interface to support elevated DPAPI operations.

```
// Encrypts data with both caller and SYSTEM context DPAPI.
//
// @param plaintext The plaintext data to encrypt.
// @param ciphertext The ciphertext of the encrypted data. It is the
//                   responsibility of the caller to free this memory using
//                   SysFreeString.
HRESULT EncryptData([in] const BSTR plaintext,
                    [out] BSTR* ciphertext);

// Decrypts data with both caller and SYSTEM context DPAPI.
//
// This will only decrypt data that was encrypted via a paired EncryptData
// call from the same user and app context.
//
// @param ciphertext The ciphertext data to decrypt.
// @param plaintext The plaintext of the decrypted data. It is the
//                  responsibility of the caller to free this memory using
//                  SysFreeString.
HRESULT DecryptData([in] const BSTR ciphertext,
                    [out] BSTR* plaintext);
```

Note: This is effectively an implementation of the proposal Extending DPAPI for Application-bound Data Protection by mikepo@. Note: this is an API that we have requested from Microsoft previously but they have declined to implement it due to resourcing/priority constraints.

Because this is just implementing an encrypt/decrypt primitive, the service can be stateless and not require any storage. This is distinct from the previous UIAccess proposal which necessitated storage with file system ACLs that the normal user could not access.

Operation within the elevated service will be to encrypt with the user's DPAPI context via impersonation, then encrypt this data with SYSTEM DPAPI. This is to avoid the service acting as an oracle for SYSTEM DPAPI.

The API will only function if the digital signature of the elevated service hosting executable matches the digital signature of the process at other end of the COM interface. This ensures that only Chrome can make the call, and also supports certificate rotation as long as the elevated service remains in sync with Chrome. As they are both installed at the same time from the same installer this seems like a reasonable assumption to make.

For callers who are running at High integrity, the service will bypass the signature checks. This is to ensure that there is an ability to import/export sensitive data e.g. other browser importing data can just elevate to obtain the key during a browser import process. This also is an explicit acknowledgement that the threat model for the service explicitly does not include that of an attacker who is able to run at elevated/High integrity.

Update (2024): It was decided in order to descope the complexity of the project to use the path of the running executable rather than the digital signature. The expansion to support digital signature is described in the companion design doc. Path validation still provides equivalent

protection against a non-admin attacker, as there are currently no attacks that authenticode-based validation would have prevented that path validation does not also prevent.

This provides a direct replacement primitive for the current EncryptStringWithDPAPI and DecryptStringWithDPAPI functions currently used inside OSCrypt for securing the OSCrypt symmetric key.

The challenges with this implementation include, how to handle early browser initialization of the OSCrypt key and data migration.

Chrome integration

Code that wishes to protect secrets uses the existing OSCrypt set of apis. This code currently is initialized early in browser startup (PreCreateMainMessageLoop) by a DPAPI decryption of the encrypted key stored in local state. This operation is assumed to be non-blocking and happens on Main thread.

Later calls to OSCrypt encrypt/decrypt are defined as thread safe and non-blocking so can be called from any thread.

Network service gets passed the OSCrypt key soon after network service initialization and before any network contexts are created. This happens in src/chrome's OnNetworkServiceCreated callback. This happens on the UI thread.

Since all callers of OSCrypt encrypt and decrypt functions therefore assume that the key has been initialized and the calls will return immediately, and can be called on any thread, this presents challenges if the initialization of the OSCrypt key is performed async rather than sync in PreCreateMainMessageLoop.

OSCrypt modifications

[WIP - this part is complex]

OSCrypt will be modified to support pluggable key encryption routines rather than being hard coded to always use DPAPI. This will be via delegate interface. This allows the existing EncryptStringWithDPAPI to be replaced by a new method that does IPC to elevated data service.

Example delegate API (async):

```
class KeyEncryptionDelegate {
  public:
   enum class CryptDelegateStatus {
     kSuccess = 0,
     kKeyNotSupported,
     kFailure,
   };
```

```
  using ResponseCallback =
      base::OnceCallback<void(CryptDelegateStatus, const std::string&)>;

  virtual ~KeyEncryptionDelegate();
  virtual CryptDelegateStatus EncryptKey(const std::string& header_value,
                                         const std::string& decrypted_key,
                                         ResponseCallback callback);
  virtual CryptDelegateStatus DecryptKey(const std::string& header_value,
                                         const std::string& encrypted_key,
                                         ResponseCallback callback);
};
```

In this delegate, each implementation will be passed key prefix as **`header_value`** (e.g. "DPAPI" for current DPAPI method) and determine whether they can decrypt this particular key. The OSCrypt code will iterate through the list of delegates until it finds one that can successfully perform the decryption.

Currently, OSCrypt is responsible for managing the local_state pref that stores the encrypted key. This could potentially be changed so the delegate implementation is responsible.

OSCrypt will need to be modified to support multiple encrypted keys, since it's likely during trials both encryption methods will need to be used concurrently.

OSCrypt will be modified to change interface from static methods to instance methods and a singleton. This work is already under way.

OSCrypt could also be modified to support Async initialization. Each caller for OSCrypt could be required to acquire an instance of OSCrypt and that this initialization might block. This adds developer costs for all current consumers of OSCrypt, and also applies platform constraints on Windows that might not equally apply on other platforms - e.g on macOS initialization is not needed at all, so making it async adds unnecessary cost.

An alternative would be to continue to initialize OSCrypt during browser startup, despite the initialization being async by nature (it involves multi-process IPC). This could either be done by simply blocking the main thread during initialization, or by triggering the IPC early during initialization, continuing chrome main thread work, then blocking future OSCrypt operations (either a FinishInit function, or the actual Encrypt/Decrypt) until the initialization has been completed.

It seems undesirable to make Encrypt/Decrypt async operations since these will never ever block. Currently these operations just fail if the key has not yet been initialized.

Data migration

Comparison of the two options

This section compares the previous UIAccess (UIA) proposal with the current Elevation Service (ES) proposal

Both proposals require that Chrome be installed system-wide. This is because for UIA the binaries must be installed in a Trusted Location[1] and for ES the service installation only occurs for a system install as it requires elevated privilege to install a service on Windows. So on this scale they are equivalent.

Both proposals have the same threat model, in that they both require the attacker to be elevated or be able to inject into Chrome binaries to obtain the secrets, so they offer of a similar level of defense against the same subset of attackers.

Because the UIA proposal runs the child process as the same user as chrome, the UIA proposal requires the UIA process to **store data itself**, protected by filesystem ACLs, since a DPAPI encrypt operation performed from within the UIA process uses the same DPAPI key as the other apps running as the same user, so provides no additional protection itself. This file storage adds complication as the user would not be able to access/delete/move/copy the file, which might cause confusion for users (depending on where it is stored in the user's profile). The ES proposal, however, requires **no additional storage** because the SYSTEM DPAPI context can be used and, by definition, that context is only available to an elevated process.

In regards to general complexity of the solution, the UIA proposal involves implementing new code in the process launcher, and also implementing a standalone mojo service inside the UIA process to support the new IPCs and tests. The ES proposal builds on a component that is already shipping with Chrome, and already has a mature test suite.

Weaknesses

An attacker could inject code into Chrome browser and call the IPC interface. It would be hard to defeat a determined attacker using this technique, and any attempts would likely just end up in an escalation which we cannot reliably win. It is likely a superior approach would be to force an attacker to use these more detectable methods, and then rely on partnerships with AV, or host-based heuristics, to detect unauthorized injection into Chrome. Since detecting code that injects into Chrome is easier for Chrome to do than just arbitrary DPAPI calls or file reads, it could also be possible for Chrome to use signals of attempts to inject or manipulate the process space to perform additional actions such as invalidating session keys, see this document for more information on this approach.

Another approach would be to continue to strengthen Chrome browser process integrity by efforts such as third party injection blocking, or continue to lobby Microsoft for additional OS-based mitigations for this type of attack.

It would still be possible to use Chrome's debugging/developer-tools interfaces to query cookies/passwords dynamically. e.g. a canonical attack that exists already is to start Chrome

[1] link

headless, or point a new instance of Chrome at a copy of the user data dir. This attack remains hard to defend against.

It would still be possible to read the memory of Chrome, and extract sensitive credentials. A partial mitigation for this would be to encrypt the components/os_crypt master key with an API such as CryptProtectMemory with CRYPTPROTECTMEMORY_SAME_PROCESS, however the sensitive data such as passwords/cookies still reside in browser memory so a determined attacker could mine for them.

Consumers of the API

It is likely the first consumer of the API will be components/os_crypt which already uses a main decryption key, encrypted with DPAPI, stored in local_state. This key is versioned with magic string `const char kEncryptionVersionPrefix[] = "v10"` so this version would be bumped to 2.0 and on next load of Chrome, the master key would be re-encrypted with a new key that would be more securely stored in the Chrome Elevated Data Service.

In the future, the Chrome Elevated Data Service could potentially be transparently layered beneath the PrefService to provide an optional 'more secure' pref storage for local state, perhaps extending `base::PrefRegistry::PrefRegistrationFlags` to expose a flag for this.

Core principle considerations

Everything we do should be aligned with and consider Chrome's core principles.

Speed

Chrome browser would have to start a new child process before any calls to the data storage service could be made. Starting a UIAccess process takes longer than a normal process as it has to be brokered through the OS to get the required Integrity Level.

We would have to measure the impact this has and how early the master os_crypt key is needed. Currently, the network process gets passed the os_crypt key after it has started so it's possible that if both processes start at the same time there could be a negligible performance impact. There would have to be synchronisation to make sure the secure service process has started before any os_crypt calls can be made.

Since the master encryption key is only retrieved once during startup, this would not affect the performance of cookie or password decryption, which already uses the in-memory key (but see above for security implications).

Security

The aim of this project is to improve the security of sensitive data on the user's machine without adding too much extra complexity, and without adding any avenues for an attacker to leverage.

It is important that the IPC interface provided by the chrome_storage.exe service is simple because it will be running at High IL so will have extra privileges that we would not want an attacker to leverage.

Stability

The chrome storage process might crash, and we would need to verify that crashpad has the required privileges to debug the process and generate memory dumps. It might not.

Simplicity

Having a simple interface of Get and Set makes the design simple to understand. We need to be careful that the executable has no risky DLL dependencies or introduces DLL planting attack vectors. We should consider using a set of tests/tools to verify that no excess dependencies (e.g. those not in the Known DLL list) are mistakenly added.

Efficiency

There will be a new binary shipped with Chrome. This will require additional memory but since it only has to service the mojo interface and call to create/read/write file from disk, this should be relatively small.

Privacy considerations

The aim of this project is to make local user data more private from an attacker who is only able to execute code at the same privilege level as Chrome.

Platform-specific considerations

This will only be available on Windows. Primarily this is because it is using a quirk of the OS to allow running privileged code with no prompt. This is not possible on other OS without a service running at that privilege all the time.

It should be noted that macOS already has superior protections against other apps running on the same system with the same privileges as Chrome, as the keychain will only unseal data to other apps via a user-mediated dialog.

Risks

This project has some data-loss risks. It is possible that if a user updates to a version of Chrome that successfully migrates their master encryption key to the secure storage would lose access to their data if for some reason the service failed to start in future, or they attempted to copy their profile data to another machine (note: copying profile data to a new machine is already unsupported), or any number of "unknown unknown" scenarios, for example their administrator applies a new policy that disables UIAccess. To mitigate against this, we could

run a milestone of Chrome that stores/retrieves some benign data in the service and measure success in UMA, before making the decision to migrate any real user data to the service.

Microsoft might consider that this is not a valid use of the UI Access flag (which is primarily for accessibility apps) and get upset with us.

Vulnerability researchers or media might misconstrue the security guarantees or threat model of this service and write bad articles about how they can bypass this protection easily (which they can, by injecting into Chrome or running as Admin) and we might get bad press.

Browsers that legitimately(?) want to import passwords/cookies from Chrome will no longer be able to do so.

Metrics

Success metrics

Hopefully we should see some movement in the malware world as a result of this, but I don't know how we will measure this. One idea is to consult with the Google TAG or TREX teams for more data.

Regression metrics

New UMA will be added to measure how many users are able to run the Chrome Elevated Data Service successfully and whether it is running as High IL, Medium IL+16, or failed to start correctly. Startup metrics will be monitored to make sure starting a new process doesn't regress anything.

Experiments

An experiment will be done to start the service without the data migration to determine how many users can successfully start the service and store and retrieve data.

Testing plan

Testing this will be hard as we do not use signed binaries on the bots. We might have to set up a self-signing certificate and install that dynamically during tests, in order to verify the functionality. However, the rest of the functionality can be tested without needing the uiAccess part to work, as the behavior is identical, but without uiAccess there is no actual protection of the process. The part that will be different is that we cannot deny Medium IL from the data file using mandatory label unless uiAccess is working.

Rollout plan

Feature will be behind a feature flag. Finch will be used to measure the perf costs and determine how many users are able to successfully get benefit from this system. However once data has

been migrated it cannot safely be turned off without data loss, so care must be taken to run the experiments wisely!

Follow-up work

It would be possible to extend the service in many ways, for example:

1. More customers e.g. it might be possible for other areas of Chrome that deal with data that we don't want trivially manipulated such as Home Page, Default Search Engine to place their data in the service, thus hindering an attacker running at the same privilege as Chrome from manipulating those values.
2. Once the first release is complete, stronger checks on the caller provenance could be implemented over time, such as thread stack examination or code integrity checks on the calling process.
3. Longer term, the protection offered by the service could be steadily improved over time, if it is proven useful and/or attackers begin to change their tools and techniques. For example, the service could be moved to a Windows Service running as LocalSystem that is installed as part of Chrome (system level install). The interfaces could remain the same so Chrome would just use whatever protection is best. A further extension would be to extend the service to have a full secure boot chain via a signed ELAM (Early Load Anti-Malware) driver to a protected User-Mode process that provides the API.
4. Given higher assurance of the calling process state, and the code could all be reused- the Chrome Elevated Data Service could also provide a basic Chrome-attestation primitive to be able to determine whether the Chrome browser is in a valid state by e.g. minting a token issued from a Google token service. This would likely have to be combined with other improvements listed above to provide high assurance (e.g. running at LocalSystem or a protected process), since this design does not cover a local attacker with High/Admin/System privilege.
5. Emit event logs for incident response and/or forensics if something unauthorised tries to call the IPC. **done.**
6. Because the master encryption key used by os_crypt will still be in-memory of the Chrome process, an attacker could simply read that key out without needing to elevate or inject any code, and use that to decrypt the files on disk. To protect against this, CryptProtectMemory with CRYPTPROTECTMEMORY_SAME_PROCESS could be used to prevent other processes from reading this memory. This might have perf implications though. **done.**
7. The service could offer an IPC to shut itself down. This could be called by the browser immediately after obtaining any required keys, to prevent any other software on the machine injecting into a running Chrome and calling the IPC.
8. VBS Enclave could be used to further secure the data, but this would need an API to be provided in the secure kernel to provide information about the VTL0 kernel, in order to provide application attestation.

(potentially bad ideas below)

9. A form of secure attention sequence (like clicking a button, or doing a remote gaia auth with security key) could be added to unlock data. This would however be a terrible UI as

it would appear on every Chrome startup, as the master key is needed early in Chrome startup.

10. (alternatively) Instead of just trusting chrome.exe implicitly, just implement an interface similar to macOS keychain whereby the user is prompted with the name of the application wanting to retrieve the data and a 'remember my decision' checkbox. If an attacker later comes along and tries to get the data then it will prompt. This still allows the case of browser import to function, with the user mediating the call. This does not prevent an attacker from just injecting into Chrome to retrieve the data.

Appendix A: Alternative UI_Access proposal

Summary

A new process called chrome_store.exe (name tbd) will be created as part of the build process. This will have UIAccess manifest and Chrome browser process will start this child early in startup using ShellExecute. The new process will be running at higher than Medium Integrity Level (IL) so cannot trivially be opened/manipulated by a normal process running at Medium. This process will expose an IPC interface to the Chrome browser process that will allow key/value storage/retrieval. The IPC will check the provenance of the other end of the caller by performing process inspection. The data will be stored in user data dir with ACLs that allow only chrome_store.exe to access it.

os_crypt will be changed to instead of storing the os_crypt master key in local_state, store it in chrome_store's storage. This can be done by upgrading the master key to new storage when it is determined chrome_store.exe is running.

Future extensions to this same design could be used to move the Chrome Elevated Data Service into a real Windows service (running as Admin) further hardening the design, but with added deployment friction. A further extension could create a fully secure pathway from EFI SecureBoot to a protected user mode service using ELAM drivers. The API in Chrome would be agnostic to how the data storage system is implemented and it could be upgraded over time.

UI Access processes

UI Access is a special privilege bit that was invented by Microsoft to allow lower privilege processes the ability to drive UI in higher privilege processes, e.g. for accessibility purposes. There's a pretty comprehensive description of it here. It's supported on Windows Vista SP1 and above.

How this works under the hood is that an application can gain this privilege if all of the following are true:

1. The uiAccess flag MUST be set to True in the application's manifest.
2. The code MUST be digitally signed.
3. The application MUST reside in a trusted location (e.g. Program Files or Windows system dir), otherwise the uiAccess flag is ignored.
4. Application MUST be launched from ShellExecute (i.e. not CreateProcess) in order to go through the UAC path via AppInfo Service[2].

Assuming these criteria are met, Windows will automatically give the running application two special characteristics:

[2] See http://securityinternals.blogspot.com/2014/03/application-information-service.html

1. The special 'UiAccess' privilege is assigned to the process. This allows the process to script other processes and bypass UIPI.
2. The integrity level (IL) is raised in order to prevent lower privilege processes from e.g. injecting into the new UiAccess process and using it as a stepping stone to unauthorised elevation.

For the second point, there's three main scenarios here that we have to consider:

1. The user is running in 'admin approval' (aka split-token) mode i.e. they are an administrator on the machine but are using UAC. This is the default configuration for most home users. In this case, the application will run at **High IL**.
2. The user is not an administrator on the machine, in which case the application will run at **Medium IL+16** (yes, the number sixteen). e.g. an enterprise/school environment where users are not admins on their own machine.
3. The user is an administrator, not in UAC/admin-approval mode, i.e. all their applications are already running at High, in which case the application runs as **High IL** because everything is running at High IL #yolo.This requires the user to have specifically disabled UAC.

We can leverage this second characteristic of raising the integrity level to provide ourselves a very lightweight 'protected process' from other processes running at Medium IL, as Medium IL processes cannot 'read up' to a higher IL. This prevents the process from being opened, debugged or the memory read/written to from other processes running at Medium.

How it works

We create a new process e.g. called ui_access_service.exe. We can link the process with the following information added to the manifest.

```
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
  <security>
    <requestedPrivileges>
      <requestedExecutionLevel level="asInvoker" uiAccess="true"/>
    </requestedPrivileges>
  </security>
</trustInfo>
```

And then sign the process with the Google LLC key (as we already do for all official binaries we ship). For those installations that are 'system-level' i.e. they install into Program Files, when launched by ShellExecute, this process will either end up at Medium+16 or High IL (see above). This means it cannot be OpenProcess or debugged from Medium IL processes such as Chrome.

For those installations where Chrome is running as a user-level install e.g. Canary or where the user did not have privilege, the application will simply not elevate at all, but will run as the same IL as Chrome. For these users there is no additional protection in place, as an attacker could just read the storage file from disk.

Chrome Elevated Data Service can implement a simple mojo interface to store/retrieve key/value pairs a bit like the current Pref Service.

```
interface SecureDataStore {
  Set(string key, mojo_base.mojom.Value value);
  Get(string key) => (mojo_base.mojom.Value result);
};
```

In fact, it could be layered below the Pref Service to provide generic storage/retrieval of any pref that is considered sensitive e.g. Default Home Page, Search Engine or anything that wished to remain less insecure than just sitting on disk.

chrome_strorage.exe will back its more secure storage with a file on disk that has Mandatory Label to deny Medium IL from reading/writing. This can be done on first run of the process, by passing the user data dir to the process (similar to how crash dir is passed to Crashpad). This will ensure that chrome.exe or any other processes running at Medium cannot trivially access the data, and would need to elevate to High. They also cannot use the same UIAccess trick, since they would need to write their binary to a 'Secure Location' which in itself needs Admin access. This data would in addition be stored with DPAPI so the data contained in the file was bound to the user/machine, retaining existing protections against offline decryption.

The ui_access_service.exe would perform a set of checks on the caller to the IPC to verify that they are, in fact, a legitimate copy of chrome and not just an attacker trying to read storage. This could range from:

1. Verify that the digital signature of the calling process matches the same one as ui_access_service.exe
2. We could do more stuff, but this could become an arms race we are unlikely to win, except if we can make it more noisy.

This is where a determined attacker could bypass the protection by e.g. injecting code into Chrome to make the IPC call. However, there are already ways for an attacker to retrieve sensitive data from Chrome e.g. using headless and javascript debugging, so the aim here is to move the attacker to a technique that is more likely to be detected by host based systems - e.g. if an attacker can no longer just run a python script to dump passwords and has to inject into Chrome, then it is possible to detect this technique with IDS, or work further on third party injection blocking which might make some headway to prevent this.

A fuller discussion on how to prevent these types of attacks is beyond the scope of this document, but some methods are expanded upon in the Follow-up work section.

Child Process Mechanics

The child process that runs as UI access has to be a different executable since the chrome.exe process cannot have the UI access bit. It also has to be signed and placed into a system folder to work correctly.

Proposal is that the new process be called uiaccess_service.exe and a CL that adds it can be found here.

How the uiaccess_service process hosts the elevated service key management is up for debate. Note: These options all assume that uiaccess_service does some basic level of attestation that the parent process or the caller is a valid chrome.exe before servicing any requests.

The reader should be aware that at a very crude level, all this process needs to do is open a file on disk, read and/or write to it, return a value, then terminate. Broccoliman "all I want to do is open a file" memes come to mind here.

In all seriousness, however, there are a number of options:

Option 1: self hosting / very basic

This option makes the uiaccess_service.exe as small and lightweight as possible, and hosts the code required to read/write the key storage and takes input/output from command line and stdout. e.g. uiaccess_service could take a command line to set the key

```
uiaccess_service.exe –set-key=<hex encoded key>
```

and another one to get key

```
uiaccess_service.exe –get-key
```

and key is output on stdout.

The advantages of this are that the binary is very small, very limited attack surface. However it does mean that a specific process launcher will need to be written in chrome/ to launch the process and allow access to the command line API. This can be done by posting a task to the launcher thread obtained by calling `GetProcessLauncherTaskRunner` on content/public/browser/child_process_launcher_utils.h.

Option 2: self hosting / mojo

This option would create a legitimate mojo service i.e. link mojo core embedder with a standalone EXE and expose a mojo api for getting/setting key. This would again require a custom launcher that is able to send the mojo invitation and create the process on the right task runner, as above. However, it has the advantage that it's using legitimate mojo IPC and so the API could be expanded more easily going forward. It would also potentially allow the interface to be implemented in e.g. browser process (in process host) for those platforms that don't support running elevated data storage, making the design more elegant.

This does have the disadvantage of increasing the binary size of uiaccess_service.exe as it will need to ship all of mojo. It also means writing the invitation/process launcher code in chrome/ somewhere, because child_process_launcher could not be used as it expects the target to implement mojom::ChildProcess mojo interface to bootstrap the process.

Option 3: self hosting but implement ChildProcess and the service.

This option would be to take the required code from content/child and implement a full child process interface, and service registry to expose the service mojo interface. It would require the exe to link with large parts of content/child content/common, and also implement the bootstrapping required for a child process.

This option adds considerable complexity and has not been prototyped. content/child contains quite a lot of deps that would certainly not be necessary to just launch this service, so it sounds like this would be considerable overkill, and also bloat binary size quite a lot.

Option 4: Child process launcher but not child process.

This is a suggestion from rockot@ which is to enlighten content's service process host interface to allow a caller to specify that the primordial interface for the child process is a mojo interface other than mojom::ChildProcess. This would mean that the uiaccess_service would implement a simple mojo embedder with invitation pipe over command-line switches that matches that of a normal child process, but would connect the elevated service interface to the invitation rather than an instance of ChildProcess. It would thus not need to pull in any of the content/child code and could be as simple as just connecting the invitation directly to the mojo service (similar to Option 2).

In content's child process host, it would be enlightened to understand that it is not connecting a mojom::ChildProcess but the specified elevated service interface. This does mean, unlike Option 2, that the existing service host interface could be used along with the existing child process launcher, meaning that code to manage the child process would not need to be written (as it would have to in Option 2).

This is similar (but different) to how mojom:CloudPrintUtility mojo interface is implemented, which is not a mojom::ChildProcess, but instead hosted in a full chrome process.

This code is all quite gnarly and could suffer from an explosion of complexity if an implementation is attempted. In addition, the cloud print service parts are soon to be deprecated so basing any new technology on the same design pattern as them might be unwise.

Option 5: New exe, same dll

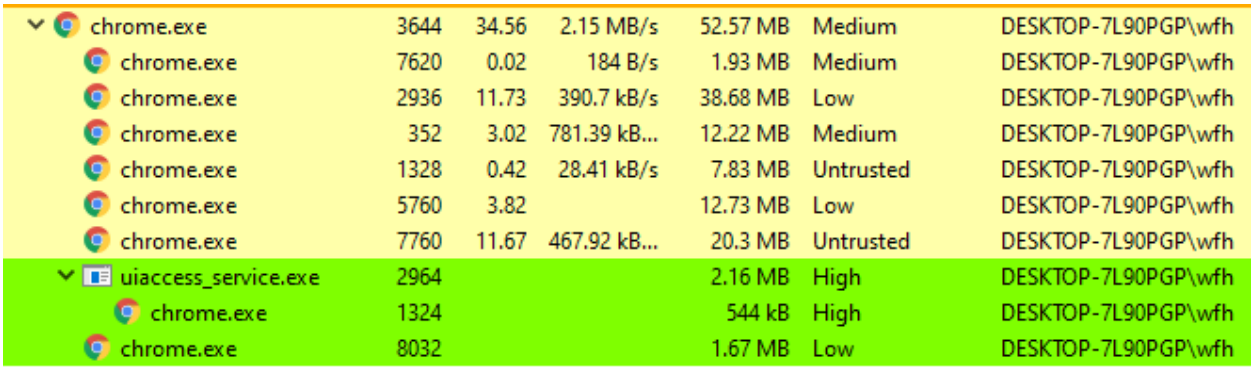
It might be possible to have a separate exe and perform enough bootstrap and then load the existing chrome.dll to launch the rest of chrome. However, chrome.exe does quite a lot of things (and is 2.5M), and duplicating this all into a new GN target might result in a large/similar binary size.

(Option 4.b) would be to create a GN target that is a literal 'copy chrome.exe to a new filename and set the UI access bit in the binary'. This clearly means shipping a second full 2.5Mb exe called uiaccess_service.exe but does make the implementation trivial (assuming any assumptions about the main exe always being called chrome.exe are not blockers). It is

possible that this extra binary would compress well in the shipped updater, but it would still use disk space. Unfortunately, while being a particularly alluring option, it would not be possible to dynamically generate this file in the installer from the existing chrome.exe since changing the UI access bit in the executable would invalidate the digital signature.

Option 6: Pass through to chrome as child of child process.

This option is different from the others in that the service itself would not be hosted/linked inside the uiaccess_service.exe but instead uiaccess_service.exe would merely be responsible for passing command line arguments through to an elevated chrome.exe running as a child process hosting the service. This option has been heavily prototyped in a CL.



chrome.exe	3644	34.56	2.15 MB/s	52.57 MB	Medium	DESKTOP-7L90PGP\wfh
chrome.exe	7620	0.02	184 B/s	1.93 MB	Medium	DESKTOP-7L90PGP\wfh
chrome.exe	2936	11.73	390.7 kB/s	38.68 MB	Low	DESKTOP-7L90PGP\wfh
chrome.exe	352	3.02	781.39 kB...	12.22 MB	Medium	DESKTOP-7L90PGP\wfh
chrome.exe	1328	0.42	28.41 kB/s	7.83 MB	Untrusted	DESKTOP-7L90PGP\wfh
chrome.exe	5760	3.82		12.73 MB	Low	DESKTOP-7L90PGP\wfh
chrome.exe	7760	11.67	467.92 kB...	20.3 MB	Untrusted	DESKTOP-7L90PGP\wfh
uiaccess_service.exe	2964			2.16 MB	High	DESKTOP-7L90PGP\wfh
chrome.exe	1324			544 kB	High	DESKTOP-7L90PGP\wfh
chrome.exe	8032			1.67 MB	Low	DESKTOP-7L90PGP\wfh

chrome.exe running at High Integrity (!!)

There are a few complexities with this option but none of them seem insurmountable.

1. Child process launcher would have to be enlightened to know that it is launching the service inside a child process called uiaccess_service.exe and not chrome.exe - this could be done via a combination of content::ChildProcessHost flags to specify this, and a new Windows implementation of ContentClient::GetChildProcessPath that understands that a particular flag means to use a different child process name. This is prototyped here. Note, this would also be needed for Options 3, 4, 5 above.
2. base::LaunchElevatedProcess would need to be enlightened to know to use the correct lpVerb when starting the process, since the verb for starting a UI access process is "" (empty string) but for an elevated process it is "runas". This is prototyped here.
3. Because of how UI access security works, uiaccess_service would have to strip the UI access token in order for the child chrome.exe process to correctly launch elevated. This is prototyped here.
4. uiaccess_service would need to perform some modification of the command line before passing to child process. There are two major parts
    a. The finch field trial handle parameters include a handle value that is passed via handle inheritance. This would need to be duplicated into the child chrome.exe process and then the command line updated. This is prototyped here.
    b. Command line parameters would need to be sanitized to make sure no malicious parameters are being passed - this is because any user on the machine can execute the uiaccess_service.exe process and it runs as High integrity, so we

need to make sure it's not possible to run chrome.exe at High integrity and unintentionally provide "privilege escalation as a service".

5. Chrome (and mojo) would need to be educated that the process hosting the mojom::ChildProcess implementation does not match the process created by the child process launcher. This might be the current stumbling block for getting this prototype fully operational as I have not yet successfully managed to get this prototype to work.

The benefit of this option is that elevated child processes are already a fully supported construct in content, and the child process would be a fully fledged child, with all support such as task manager recognition, child process termination metrics etc. Clearly crashpad would not be able to debug the process though, so any crashes in the process might not be measurable, but this is an issue already for elevated child processes.

The other concerns with this option are that there is potentially a very large amount of code running elevated, since an entire chrome.exe is now running at High integrity hosting the service. It is also impossible for the user, or chrome.exe (running medium) to perform any operations on the process (by definition, since it's elevated) and that includes crashpad and process termination. It might be possible for the process to enter into a state where it is unkillable e.g. if it hangs unexpectedly, and it cannot be terminated by the user or crashpad. This is undesirable.

The desire to design for "attack surface reduction' and 'keep things simple' certainly incentivises choosing one of the earlier options - limiting the amount of code that is running elevated is a good security practice here. This also limits the risk of an attacker finding out a way to convert the ability to run chrome.exe at High integrity into a very nasty privilege escalation.

However, the desire to make things as close to existing code as possible (note: we already support running services elevated in a chrome.exe e.g. the RemovableStorageWriter so it's not unprecedented to have this amount of code running elevated) certainly promotes the choice of one of the latter options, indeed it seems to make most sense to expose a mojo interface to the service if only to make expansion of the APIs offered by the elevated process easier going forward.

Analysis

My current thinking on this is that **Option 1** is out because it's too limiting for future expansion, and not using mojo at all for IPC seems like a Bad Idea. **Option 3** also seems unworkable, for pulling in all of content/child and yet not making the process chrome.exe. **Option 5** is very simple implementation but requires shipping 2.5Mb of copy of chrome.exe so might be precluded simply based on that (unless it can be determined it does not bloat installer size because of compression).

Option 4 has a certain elegance, but seems like a very large amount of code and support to add into content's service manager layer for one consumer.

This leaves **Option 2** and **Option 6**. **Option 6** has considerable merit but I have not yet managed to get it to work and I don't know why not. It also means hosting a very large amount of code running elevated and there are attack surface / complexity concerns here. It does have the

considerable merit of fitting very well into content's existing process architecture, with minimal changes though. I believe it is worth continuing the investigation into this to try and work out root cause of why it's not working, since by all indications it should.

Option 2 only remains, which is to embed mojo with the new process, and send invitation from chrome/ code with a custom launcher. This is similar to approaches taken with other bespoke child processes such as chrome cleaner and recovery component, excepting that the uiaccess_service would IPC using mojo (which seems desirable)

At time of writing, I am thinking that **Option 2** has the right balance of simplicity, security, binary size, and standardization/expandability (since it uses mojo for IPC). It would be good to know exactly why **Option 6** is not working though.