info@apriorit.com
www.apriorit.com

Case study

Developing a Custom Driver Solution for Blocking USB Devices

Keeping an eye on USB connections is an important part of many cybersecurity practices. This is why our client decided to enhance their enterprise product by adding functionality for blocking restricted USB devices.

To achieve this goal, the client searched for a professional team with experience in custom driver development. After researching the market, they decided to go with Apriorit.

Explore this case study to find out how we helped our client deliver a driver for managing ports and what challenges we faced along the way.

The client

Our client is a leading US-based provider of advanced encryption solutions for protecting sensitive data and intellectual property. They offer a complete suite of encrypted hardware products, virtual drives, and centralized management platforms.

The challenge

The client requested a driver for managing USB ports on computers and blocking USB devices according to a set of rules.

The client challenged us to develop a driver for blocking USB devices that would:

- Block a wide range of USB devices based on sets of rules
- Support different versions of Windows and macOS
- Keep a computer's basic functionality working in case of an incorrect combination of rules — for example, after blocking some internal USB or USB-like connections — until the support team changes the rules
- Be compatible with the client's other solutions, as it was supposed to be one component of the client's enterprise software product.

4 main challenges

Block USB devices according to a set of rules	Provide driver support for Windows and macOS
Keep the computer working if rules contradict each other	Ensure driver compatibility with the client's other solutions

The main challenge was to deliver a driver that at the same time must be secure, prevent unauthorized use of USB devices, and guarantee the work of the end point with any combination of rules. The client requested a driver for managing USB ports on computers and blocking USB devices according to a set of rules.

Our approach

During the discovery stage, we assessed the client's initial requirements and discussed the scope of work, iterations, and time frames for each iteration. Apriorit's dedicated team for building a solution for USB port blocking included a developer, quality assurance engineer, business analyst, and project manager.

Since the scope of work was clear and only a few questions arose during development, we chose weekly reports over regular calls as a communication strategy. This was enough for our client to stay abreast of current tasks and upcoming issues.

When we moved to adjusting the developed driver and adding new features, our team started working with the client's Jira. This ensured swift work with arising issues and feature requests and allowed our client to communicate with the team directly in the comments on Jira tickets instead of relying on emails.

The result

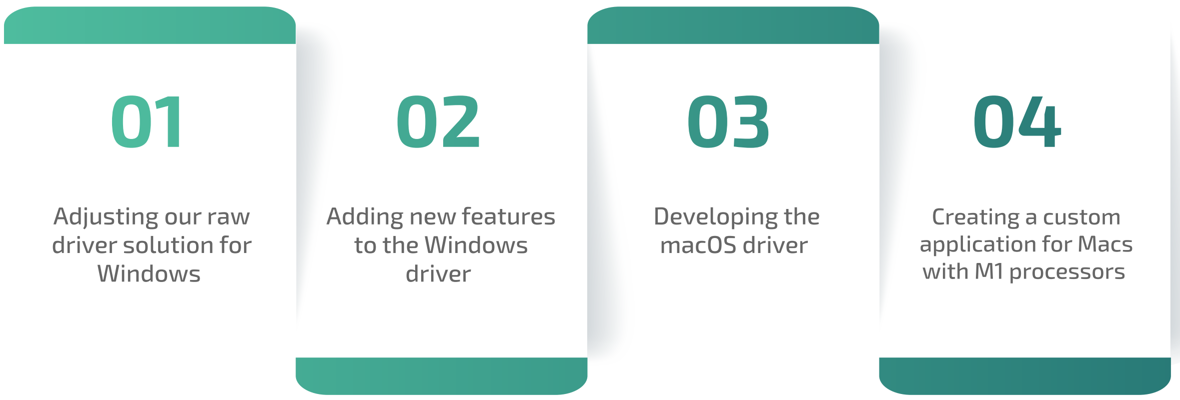
The Apriorit team successfully delivered an efficient driver for blocking USB devices. To make it meet all of the client's requirements, we made necessary adjustments and thoroughly tested each feature before its release.

To ensure the driver's work on Windows, we used our existing raw solution and modified it. For macOS, our team developed a solution from scratch.

How we did it

To build a solution to block USB ports and help our client deliver the desired product, we thoroughly planned our work and divided the entire process into four main stages.

KEY PROJECT STAGES


01
Adjusting our raw driver solution for Windows
02
Adding new features to the Windows driver
03
Developing the macOS driver
04
Creating a custom application for Macs with M1 processors

Stage 1. Adjusting our raw driver solution for Windows

At Apriorit, we already had a raw solution — a USB blocking SDK — that we developed earlier and that needed to be adjusted for the client's needs. The raw driver was written in C++ for Linux, so we had to port it to Windows.

Initially, our solution was managed via blocklists and allowlists:

- The **blocklist** contains devices that must be blocked. Users can instantly block all printers, mass storage devices, etc. instead of blocking such devices one by one. All devices that are not added to the blocklist are allowed.

- The **allowlist** consists of two parts. The user-defined allowlist permits users, for example, to block access to all printers except a specific one. The internal allowlist includes devices that must be allowed to guarantee the computer's functioning.

Both lists are stored in the registry. An important note is that if rules are changed, they only apply to newly connected devices. After we demonstrated to our client how our raw solution worked on a demo application, we agreed to introduce the following modifications:

- Change the blocklist and allowlist update and storage mechanisms to match the requirements of the client's larger system

- Change list formats to meet the client's rule requirements

- Add a driver installer

- Sign the driver with the client's digital signature

- Add tray notifications to let end users know that a device has been blocked

Once we introduced all the changes, we also prepared documentation and tested the final driver before sending it to the client. After the successful product release, we moved to support activities.

Stage 2. Adding new features to the Windows driver

Our first task during this stage was to add support for Windows 10 to our driver. Then, the client requested several new changes, in response to which we did the following:

- Enhanced the driver's compatibility with the client's system by adding a TypeScript wrapper and a native Node.js module

- Improved our driver's flexibility by adding the ability for read-only access to devices apart from blocking and not blocking

- Made device management more user-friendly for administrators by introducing changes to the driver that made it possible to get more information about detected USB devices

- Enhanced driver functionality and provided better protection of end-user data by adding support for UASP storage devices and VDI environments: Hyper-V, VMware, and Citrix XenApp

- Thoroughly tested all changes before presenting them to the client and releasing the driver

After the successful Windows driver release, our team moved to implementing USB device blocking functionality for macOS.

Stage 3. Developing the macOS driver

During our research, we found that existing third-party solutions could not fully meet all of the client's requirements, so we started developing a custom driver for managing USB ports for macOS.

To implement the required functionality, we decided to write a **kernel extension (kext)** in C++. Our proposed driver solution works as follows:

1. When a user plugs in a device, the kernel looks for dexts (DriverKit extensions) and kexts to match the device.

2. Found drivers are loaded following the probe score set for them by developers.

3. Our kext is attached after the system kexts. Once attached, our kext checks if the plugged-in device must be blocked according to the defined rules.

4. If the device must be blocked, the kext detaches those matched USB drivers that were attached before.

5. After this, the device becomes invisible to the system, as there are no longer USB drivers to support its work. Thus, the device is effectively blocked.

Moreover, even if a user tries to attach blocked drivers manually to make the device available, our kext will detect it and continue detaching them.

If rules suggest that a certain device must be read-only instead of being blocked entirely, our kext will act slightly differently. In this case, it mounts the device in read-only mode, which doesn't affect driver visibility to the system and allows users to still work with the device. But having a high probe score, our kext prevents any other extension from changing the read-only setting for a device, preventing data from being copied to it.

Once we developed the driver, we prepared a demonstration application written in Swift that allowed both our QAs and our client's QAs to check the driver behavior with different sets of rules and devices.

According to the client's requirements, we ensured that the driver works on macOS versions up to 10.11.x. After its successful release, we started adding support for newer versions of macOS.

Stage 4. Creating a custom application for Macs with M1 processors

In 2020, Apple released the M1 processor, which has limited possibilities for kexts. Since the existing driver could no longer be used for computers with M1 processors, we had to create a new kextless application.

After some research, we developed a new custom solution that was a macOS system daemon wrapped in the bundle structure and signed with **endpoint security entitlement** with the following key characteristics:

- The application receives device blocking rules from a locally stored JSON file.
- The application can establish a connection with the client application via XPC to send/receive blocking commands and event logs.

We managed to deliver an application that completely operates in user mode and at the same time provides all the functionality of the kext-based solution.

Once the kextless solution was successfully released, our team moved to implementing new features. We added support for a wider range of USB devices and made general driver behavior more user-friendly, such as by adding pop-up notifications explaining to the end user why a USB device has been blocked.

We keep checking each macOS update to make sure our application continues to work correctly.

Challenges and solutions

During the custom USB device blocking solution development and adjustment processes, our team faced a few challenges. The most crucial were the following:

3 key project challenges

- **Developing a kextless application**
- **Supporting various USB devices**
- **Avoiding blocking essential system functionality**

1. Developing a kextless application

This was the biggest technical challenge faced by our team during this project. We conducted in-depth research looking for possible third-party solutions and didn't find anything that could be used. Then, our team moved to preparing a proof of concept (PoC) and looking for the possibility to block the device or make it read-only from user mode. The proposed solution worked flawlessly and allowed the client to achieve their goal, even on Macs with M1 processors.

2. Supporting various USB devices

There are a great variety of USB devices out there, and we wanted to be sure that our solution would work smoothly with most of them. To do that, we:

- Tested the most important groups of devices, such as storage devices, smartphones, audio devices, and video devices

- Asked Apriorit employees to share unique USB devices like USB shredders, USB fans, and USB kettles, and checked them as well

This allowed us to make sure our driver can detect and block even the most exotic devices of lesser-known vendors.

3. Avoiding blocking essential system functionality

When developing device blocking solutions, there's always a risk that they could block some of the computer's essential functionality and make the device unusable. To avoid that, we implemented default allowlists that allow some devices to never be blocked irrespective of the administrator's settings. This guarantees that even if someone messes with settings or sets them carelessly, the support team will always be able to fix such an issue and make the endpoint work correctly.

The impact

The developed solution helped our client launch their software product with the ability to block ports for USB devices and allow only client-approved devices. Thus, the driver created by the Apriorit team increased both the value of the client's product and the client's competitive advantage.

We continue supporting the client's USB blocking solution, adjusting it to the ever-changing world and adding new features at the client's request.

Ready to develop a custom driver solution?
Contact Apriorit to leverage our development skills and expertise!