A Privacy First Home Surveillance System: Local NVR Design and Implementation

Hayden Phifer

Executive Summary

Commercial home camera systems are marketed as safety products, but they depend heavily on vendor cloud infrastructure that I do not control. High profile incidents involving employees misusing access and large-scale breaches have made me deeply uncomfortable with the idea of putting live video from my apartment on remote servers. At the same time, my partner and I will be living apart for periods of time, and she very reasonably wants cameras so she can check on the home and pets when she is away. This project grew out of that tension. I wanted the reassurance of cameras without the surveillance culture that comes with a typical cloud NVR.

In this MSIS capstone I designed and implemented a privacy first home surveillance system that keeps all video local and only allows remote access through a VPN that I control. The system uses a Reolink E1 Pro IP camera that exposes an RTSP stream, a repurposed Alienware X51 R3 desktop running Windows 10 IoT Enterprise as the recorder, and a UniFi Dream Router 7 as the security gateway. OBS Studio runs on the Alienware box to ingest the RTSP stream and record continuous video into timestamped segments. Cobian Reflector runs as a Windows service and creates encrypted backup copies of those segments on a BitLocker protected volume so that a single disk failure or theft will not automatically expose the archive. The UniFi Dream Router enforces strict network segmentation, blocks the camera from reaching the Internet, and terminates WireGuard VPN sessions for remote viewing.

The network design treats the camera as untrusted by default. It resides on its own VLAN, and firewall rules deny outbound traffic from that network to the Internet. During the project I captured and analyzed UniFi firewall logs and saw the camera repeatedly attempting to contact a large number of external IP addresses. All of those attempts were blocked, confirming that the router was actually enforcing the local only posture that the project requires. At the same time, WireGuard peers on my phone and laptop can join the home network as if they were on site and reach the recorder through normal routing, so I can securely view live or recent footage over an encrypted tunnel without exposing any web portals or RTSP endpoints to the public Internet.

Getting this system stable required more than just wiring things together. Long running OBS sessions initially froze when the RTSP stream stalled, and my first Cobian configurations tried to touch files that were still being written. Through iterative testing I moved the RTSP path to TCP, added reconnect and timeout options, adjusted recording destinations, and refined backup jobs so that recordings and backups could run side by side without interfering with each other. With those changes in place, the system can continuously record from the camera, retain at least a full day of footage with practical headroom toward roughly 12-14 days, and maintain a parallel encrypted backup set on a separate logical volume.

By the end of the semester, the project met its core goals. I can operate a local only surveillance system that does not depend on a vendor cloud, which keeps footage under my control, and that still allows me to check on my home from anywhere using a modern VPN. The work also sharpened my skills in network segmentation, firewall policy design, VPN configuration, backup and retention planning, and long running application tuning. There is room to extend this platform with additional cameras, better storage, and more automation, but even in its current form it is a working example of how to build a home surveillance system that takes privacy seriously instead of treating it as an afterthought.

1. Introduction and Personal Context

This project sits at the intersection of my academic work, my military experience, and a very personal concern about how much of our lives we hand over to other people's computers. ISMN 7980 is the capstone course for my Master of Science in Information Systems Management. The expectation is not just that students build something that works, but that we can design it, implement it, test it, and then explain it clearly enough that someone who was not in the room for every troubleshooting session can still understand what we did and why it matters. This system is the technical and narrative product of my studies.

My path into this project was not a straight line from "I like networks" to "I will build an NVR." It took me six years to finish my bachelor's degree because I changed majors multiple times and eventually ran into the reality that I could not afford to stay in school without a new plan. I enlisted in the Army National Guard to pay for college, finished my degree in aviation management, and then realized I was not excited about the long-term career paths in that field. What I was actually interested in, and what I kept coming back to on my own time, was information systems and cybersecurity. That is what led me into the MSIS program and into work with a cyber protection team, and it is also what shaped how I think about "simple" consumer technologies like cameras and routers.

The immediate spark for this project was much smaller and more personal. My partner and I have spent most of our relationship juggling different schedules, training commitments, and the reality that there would be stretches of time where one of us was away and the other was home alone. At one point she asked if we could install cameras in the apartment so she could check on the place and the pets when she was gone. Intuitively that made sense, but I could not ignore everything I had been reading and learning about commercial camera platforms. There is a steady trickle of stories about vendor employees spying on customers, about internal tools being abused, and about breach after breach where live streams or archives leak out in ways that are very hard to undo. The idea of pointing a camera at our living space and then streaming it to someone else's cloud felt wrong.

That tension is what pushed this idea from a late-night rant into a real project. I wanted the same comfort and visibility that a normal cloud camera system would give her, but I wanted to get it without asking a third party to warehouse video from our home. ISMN 7980 gave me the time and structure to turn that into a concrete design problem. The rest of this report explains how I translated that motivation into requirements, architecture, implementation, and testing, and how I ended up with a system that actually runs in my apartment today.

The remainder of this report is organized as follows. Section 2 defines the problem statement and privacy motivation in more detail. Section 3 describes the requirements and stakeholders, including the user stories and acceptance criteria that guided the work. Section 4 introduces the hardware platform, and Section 5 explains the software and network design on top of it. Section 6 walks through the implementation process and timeline, and Section 7 summarizes testing, troubleshooting, and results. Section 8 presents a structured risk analysis, Section 9 reviews major design alternatives that were rejected, and Section 10 reflects on the key struggles and lessons learned. Section 11 concludes with future enhancement ideas and the relationship between this written report and the accompanying slide deck.

2. Problem Statement and Privacy Motivation

From a purely technical perspective, a home surveillance system is straightforward. You put a camera in a room, you connect it to a network, and you send its video to a recorder. The problem is that most commercial systems do not stop there. They send your video to vendor infrastructure, they maintain their own authentication and storage layers, and they give employees and contractors levels of access that are difficult for end users to reason about. Over the last several years there have been public cases where employees at camera companies used that access to spy on customers, and there have been breaches

where large numbers of camera feeds or archived clips were exposed to people who were never supposed to see them. Once that happens, you cannot “unstream” your living room.

My threat model for this project is shaped by that reality. I am not primarily worried about an extremely sophisticated state actor targeting my apartment. Instead, I am worried about the much more mundane, but much more common, risks that come from plugging into someone else’s ecosystem. If my cameras are tied to a cloud account, then anyone who can access that account, or the systems behind it, can potentially watch my home. If my cameras are designed to phone home constantly, then even if I am careful with credentials, my traffic patterns and device behavior can still leak to third parties that do not need to know anything about me. I do not want vendor employees, contractors, or random attackers trawling through customer data to drop into a live feed of my apartment.

The Reolink E1 Pro that I used in this project is a good example of why this matters. When I first connected it to an isolated network and looked at the UniFi firewall logs, I saw it trying to contact hundreds of external IP addresses over a relatively short period. None of those attempts succeeded, because my firewall policy denied outbound traffic from the camera network to the WAN. If that policy had not existed, the camera would have been quietly chatting with a significant slice of the public network without me ever seeing it. In a typical consumer setup, that behavior is invisible. The default ISP router does not advertise it, and the vendor app certainly does not frame it as “this device is constantly reaching out to external servers.”

Given that context, the problem statement for this project is simple to say and harder to implement. I want a home surveillance system that gives me the benefits of cameras without outsourcing trust to a cloud provider. More concretely, I want to be able to record and store at least a full day of video locally, with room to extend that window as my storage allows. I want to keep all of that footage on hardware that I control in my home, protected by my own operating system and disk encryption choices rather than a vendor’s opaque backend. I want to be able to view a live feed, and possibly very recent clips, from my phone when I am away, but only after I have joined my home network through a VPN that I operate. I do not want to expose web portals, RTSP endpoints, or vendor relay channels to the open Internet.

Those requirements define the rest of the project. They rule out a whole class of off-the-shelf cloud camera kits because the way those systems work is fundamentally at odds with the privacy posture I am aiming for. They push the design toward self-hosted software, toward network segmentation, and toward a model where my router and recorder enforce the rules instead of trusting that someone else will behave well on my behalf. The following

sections describe how I broke those goals down into specific functional and nonfunctional requirements and how I designed a system that meets them in practice.

3. Requirements and Stakeholders

By the time I committed this idea to my project proposal, the system had moved from a vague “privacy friendly camera setup” to something with fairly concrete requirements. Those requirements grew directly out of the threat model and personal context described in the previous sections, but they had to be precise enough that I could tell whether the system actually met them by the end of the semester. In this section I describe who the system is really for, what it is supposed to do, what constraints it must respect, and how I translated those ideas into a small set of user stories and acceptance criteria.

3.1 Stakeholders and Roles

In a typical commercial deployment, the primary “customer” is whoever buys the camera and downloads the app, and the secondary stakeholder is the vendor who operates the cloud side of the system. In this project I deliberately removed the vendor from that picture. The system is built and administered by me, runs on hardware that I own, and lives on a network that I control. I am both the system owner and the primary day to day user.

There is still a human story behind the design. My partner’s desire for cameras is what pushed me to explore this space in the first place, and part of the motivation is giving her a way to check on the apartment and the pets without asking her to trust a vendor’s servers. In the formal requirements, however, I treat her as a future secondary user rather than as the explicit “customer.” That keeps the system focused: if it is safe and usable enough for me to run every day, then it can later be extended to grant her access through the same secure mechanisms.

Other stakeholders exist at the edges. Anyone who visits the apartment is affected by the presence of cameras, even if they never see the footage. Law enforcement becomes a stakeholder in the unlikely but serious case where recorded video is requested as evidence. These roles influence how I think about retention, encryption, and physical security, but for the purpose of this project the primary requirements revolve around my own administrative role and my need for a reliable, private surveillance platform.

3.2 Functional Requirements

Functionally, the system has to behave like a real surveillance system, not just a proof of concept. At a minimum, it must continuously record video from the Reolink E1 Pro and store enough history that I can go back to and review events after the fact. Early drafts of

the project documents set a clear baseline: at least twenty-four hours of retained footage, with a strong preference for several days if storage allows. That requirement comes from a simple scenario. If something happens at home while I am away, I may not find out immediately. I need enough of a window to come back, review what happened, and, if necessary, preserve clips for law enforcement or insurance.

The second functional requirement is live viewing. I need to be able to see current feed from the camera when I am away from home. The critical distinction is how that access is granted. I was not willing to rely on a vendor app or public web portal. Instead, the system has to deliver live video only after a remote device joins my home network through an encrypted VPN that I operate. That means the functional requirement is not just "view live video" but "view live video as if I were on the LAN, without exposing the recorder or camera directly to the Internet."

A third functional requirement concerns playback. On the local network, I should be able to open recent recordings on the Alienware recorder and scrub through incidents without special tools beyond what a normal user could reasonably learn. Remotely, I originally sketched a stricter requirement that any playback over VPN should be limited to a very short window into the past, on the order of a few minutes, to reduce the risk of someone casually browsing an entire archive. In practice that turned into more of a design goal than a fully enforced technical constraint, but it still informed how I thought about retention and access.

3.3 Nonfunctional Requirements

The nonfunctional requirements are where this project diverges most sharply from a typical consumer camera deployment. The first and most important is privacy. All footage must remain on self-hosted hardware in my home. The camera network must not have unrestricted access to the Internet, and the recorder must not be reachable from the public side of the router except through a VPN endpoint that I control. Any stored copies of footage must be encrypted so that theft of a disk does not automatically reveal the contents.

Cost is the second major constraint. I set out from the beginning to reuse my existing Alienware PC and to select a camera and router that were within a student budget. That means the design has to work within the limits of a single recording host and commodity storage rather than assuming redundant arrays or dedicated appliances. It also means favoring tools like OBS and Cobian that are either free or relatively low cost.

Reliability is the third nonfunctional requirement. The system does not have to meet five nines of availability, but it does need to run unattended for long stretches without constant

manual intervention. If the camera or network glitches, the recorder should be able to recover and resume recording. If the host reboots, the recording and backup services should come back up on their own. For a system that is intended to provide reassurance when I am away, constant manual intervention would defeat the purpose.

Finally, there is a usability requirement tuned to reality: I am the only administrator. The system must be understandable and maintainable by a single person who also has other responsibilities. Configuration should be scriptable or at least well documented enough that I can rebuild it after a failure without recreating weeks of trial and error from memory.

3.4 Acceptance Criteria and User Stories

To know whether I had actually met these requirements, I translated them into a few simple user stories and acceptance criteria. One story captures the remote access behavior: as the system owner, I want to open a live view of the camera from my phone while I am away, but only after I have connected to my home network over VPN, so that no public endpoints expose my video feed. Another story focuses on incident response: as the system owner, I want to review footage from any point in the last twenty-four hours and export a clip if necessary, so that I can understand and preserve what happened without having to sift through logs or raw files by hand.

These stories convert directly into acceptance tests. At the end of the project, I should be able to show that the system can continuously record at least a full day of video without filling the disk, that I can remotely view the live feed only when connected through WireGuard, and that the camera's attempts to contact the Internet are blocked by the firewall. I should be able to demonstrate that recordings exist in both the primary OBS directory and the encrypted backup vault, and that losing a single disk or rebooting the host does not permanently break the system. Later sections of this report return to these criteria and evaluate how closely the final implementation matches the intentions laid out here. Taken together, these user stories and acceptance criteria serve as the requirements documentation for the project in place of more formal customer interviews or surveys.

4. Hardware Design and Platform

The hardware for this project grew out of three constraints that never really changed: I wanted strong privacy guarantees, I did not want to buy a rack of new gear, and I wanted to reuse hardware I already owned wherever possible. The final platform is built around a single RTSP capable IP camera, an older Alienware desktop that I repurposed as an NVR, and a UniFi Dream Router 7 that handles both network isolation and VPN access. Together

these pieces give me continuous local recording and encrypted backup without handing any footage to a cloud provider.

4.1 Camera Selection: Reolink E1 Pro

The starting point was choosing a camera that would cooperate with a self-hosted recorder. I ended up with the Reolink E1 Pro, which is a small indoor pan tilt IP camera that still exposes a straightforward RTSP stream. That RTSP support is the main reason I chose it. Many consumer cameras tunnel everything through a vendor app and cloud service, which makes them essentially unusable for a project where I want my recorder to pull a standard stream on my own network.

Image quality mattered, but I did not need a cinema camera for a small apartment; the Reolink E1 Pro provides enough detail to identify people and events without generating bitrates that would overwhelm a ten-year-old PC and a mechanical hard drive. Just as importantly, the camera can be run without ever creating a Reolink cloud account. In my final configuration, the device lives on an isolated network segment with firewall rules that prevent it from talking to the Internet at all.

The UniFi firewall logs made this choice feel justified. When I later captured logs from the camera network, I saw the E1 Pro trying to contact hundreds of external IP addresses in a fairly short window. Those attempts were blocked by my rules, but the volume of traffic was a good reminder that even a single consumer camera will happily chat with large parts of the Internet if you let it. That behavior would be invisible on a typical home router. In my design, it became evidence that the camera needed to be treated as untrusted and fenced in.

4.2 NVR Host: Alienware X51 R3

For the recorder, I did not buy a dedicated NVR appliance. Instead, I pressed an older Alienware X51 R3 desktop back into service. The machine is roughly a decade old, with an Intel Core i7 class processor, 16 gigabytes of RAM, a solid-state drive for the operating system, and a larger spinning hard disk for bulk storage. On paper, this is gaming hardware from another era. In practice, for a single high-definition camera, it is more than enough.

Reusing this PC let me avoid the cost of a new box and gave me full control over the operating system and security posture. During the project I upgraded it from Windows 10 Pro to Windows 10 IoT Enterprise so that it would continue to receive long term security updates beyond the standard desktop support window. That change also nudged me toward stripping out unnecessary services and background applications. For a machine

that will sit in a corner and quietly record video for years, fewer moving parts at the operating system level is a good thing.

In the final design, the Alienware machine wears several hats. It runs OBS Studio as the primary recorder, pulling the RTSP stream from the camera and writing segmented recording files to disk. It runs Cobian Reflector as a Windows service that creates encrypted backups of those recordings on a secondary volume. It also acts as the main endpoint for remote viewing sessions when I connect over VPN from my phone or laptop. Because all of this work is centered around a single stream, the system load is surprisingly modest. Under recording load CPU utilization stays well below saturation and disk activity is mostly sequential writes, which this hardware handles comfortably.

4.3 Router Choice: UniFi Dream Router 7

The router is where the project's privacy goals actually get enforced. Early on I experimented with or at least seriously considered other options, including the default ISP router and some consumer grade ASUS gear. Those devices could route packets, but they made it awkward to create a cleanly isolated camera network, and their logging and VPN capabilities were limited. When I realized how much time I was about to spend fighting the network, I decided to standardize on UniFi Dream Router 7.

The UDR gave me three things I needed. First, it made network segmentation straightforward. I could put the camera on its own VLAN, treat that segment as untrusted, and strictly control what traffic flows in and out. Second, it gave me good visibility into firewall behavior. The per device firewall rules and logs let me verify, rather than just assume, that the camera's attempts to contact external IPs were being dropped. Seeing hundreds of blocked outbound connections from a single camera in the logs did more to reinforce my distrust of cloud centric camera ecosystems than any marketing brochure ever could. Third, the router has integrated WireGuard VPN support. That allowed me to build a clean remote access path where my phone joins my home network as a proper peer, instead of punching random holes in the firewall or relying on a vendor relay service.

In the final topology the UniFi Dream Router sits at the center of everything. It separates the Internet, my main home network, and the camera segment. All camera traffic to and from the Alienware recorder crosses that boundary, where firewall and VPN policies can be enforced consistently.

4.4 Storage Layout and Estimated Retention

The storage layout on the recorder reflects the same "reuse what I have but do it carefully" mindset. The Alienware box has a solid-state drive that holds Windows and applications,

and a larger hard disk that I use for long term storage of video and backup data. OBS writes its primary recording files to the hard disk in a dated folder structure that makes it easy to check what is available and how quickly space is being consumed. Cobian Reflector writes encrypted backup sets to a dedicated folder that lives on a BitLocker protected volume so that someone who steals the machine does not get an unencrypted video archive as a bonus.

To understand how long I could keep footage online, I did rough retention calculations based on my encoding settings and disk capacity. Using the resolution and bitrate I settled on, the math and my slide estimates showed that a single continuous camera stream could be kept for roughly 12-14 days before the disk would fill and older files would need to be overwritten or moved. That estimate assumes the bulk of the hard disk is available for video rather than other data.

In practice, the actual retention window depends on several moving pieces. OBS can be configured to delete or retain older segments in different ways. Cobian's schedule and the number of full backup sets I keep will also consume space over time. The system is flexible enough that I can adjust those knobs later. For purposes of this course, the requirement was to reliably retain at least twenty-four hours of footage with room to grow for several days. The hardware platform, combined with conservative recording settings and encrypted backups, comfortably met that requirement.

5. Software and Network Design

The hardware choices gave me a solid foundation, but the real behavior of the system is defined by the software and network configuration that sits on top of them. In this project, the core stack consists of OBS Studio for capture and recording, Cobian Reflector for encrypted backups, UniFi OS on the Dream Router for segmentation and firewalling, and WireGuard for remote access. All of that is glued together by a fairly opinionated network design that treats the camera network as untrusted and routes everything through the recorder and VPN.

5.1 Overview of the Software Stack

From the beginning, I knew I wanted to avoid a black box NVR that hides its behavior behind a glossy web interface. Instead, I built the system out of tools that are either open source or at least transparent enough that I can understand what they are doing. OBS Studio acts as the main recorder and lives on the Alienware PC. Cobian Reflector runs on the same host as a Windows service and writes encrypted backup copies of recordings to a BitLocker protected volume. The UniFi Dream Router provides a management interface for VLANs,

firewall rules, and WireGuard, and its logs are my main source of truth about what the camera is trying to do on the network.

All of these pieces are configured with the same two themes in mind. First, they assume failure is normal. Networks glitch, processes crash, and drives fill up. Whenever possible, I wanted the system to recover without me having to log in and nurse it back to health. Second, they assume the camera and the outside world are hostile. The camera is never allowed to talk freely to the Internet, and no remote device is allowed to talk to the recorder without going through a VPN that I control.

At a concrete level, the tech stack for this project consists of Windows 10 IoT Enterprise running on the Alienware X51 R3 recorder, OBS Studio for capture and encoding of the RTSP stream from the Reolink E1 Pro camera, Cobian Reflector for encrypted backups on a BitLocker protected volume, and UniFi OS on the Dream Router for VLANs, firewall rules, and WireGuard VPN. These components, combined with the Reolink firmware on the camera itself, make up the full set of software and services that the system depends on in normal operation.

5.2 Network Topology and Segmentation

At a high level, the network is split into three domains: the public Internet, my main home LAN, and a dedicated surveillance network that contains the Reolink camera. The UniFi Dream Router sits in the middle and uses VLANs and firewall rules to keep those domains from bleeding into each other.

The camera lives on its own VLAN with an address range that only the router and the Alienware recorder can reach. From the perspective of the camera, the rest of the Internet simply does not exist. Any attempt it makes to contact an external IP is caught by a firewall rule that denies outbound traffic from that VLAN to the WAN. The network log analysis session I did partway through the project confirmed that these rules are doing real work. In a relatively short log window, the camera attempted to contact more than two hundred distinct external addresses, and the firewall blocked every single one. That data made it truly clear that a normal consumer deployment, where the default router allows outbound traffic, would quietly leak a lot of information to vendor infrastructure and content delivery networks.

The Alienware recorder sits on my main LAN but has explicit firewall rules that allow it to talk to the camera VLAN on the ports that OBS needs for RTSP. In the other direction, devices on the main LAN cannot initiate connections into the camera network unless they pass through the same controlled path. When I connect remotely over WireGuard, the VPN

peers join the main LAN and reach the recorder through ordinary routing, exactly as if I were at home.

5.3 OBS Studio Configuration and Role

OBS Studio is usually thought of as streaming software, but in this project, it behaves more like a general-purpose NVR focused on a single stream. I configured OBS to subscribe to the camera's RTSP feed and write continuous recordings to disk in timestamped segments. That segmenting behavior is important. It makes it easier to manage retention and also reduces the damage if something goes wrong during a recording window.

Getting OBS stable for multi day recording took some work. In my early tests, long running sessions would freeze without warning. The preview would stop updating, the interface would become sluggish, and the recording would silently die. After digging into logs and configuration details, the pattern became clear. By default, the RTSP path was using UDP without any reconnect or timeout settings. When the camera or network glitched, the RTSP input would stall. The encoder thread in OBS would block waiting for frames that never arrived, and the entire application would follow.

The final configuration takes a more defensive approach. The RTSP connection is forced to use TCP, which is slightly more forgiving of brief network interruptions. I added explicit reconnect and timeout flags so that if the stream drops, OBS tries to reestablish it instead of hanging forever. The recordings are written to a fast local volume in a folder tree organized by date, which keeps disk I/O mostly sequential. Together, these changes dramatically reduced the chance of a permanent freeze. If the camera or router hiccups, OBS now has a much better chance of recovering on its own and continuing to record.

5.4 Cobian Reflector Backup and Retention

While OBS handles primary capture, it is not a backup system. If the recording folder lives on a single disk and that disk fails or is stolen, all of the footage disappears with it. To address that risk, I used Cobian Reflector as a lightweight backup engine running on the same host.

Cobian is configured to run as a service under a dedicated account so that backups can proceed even if no one is logged in. I created tasks that point at the directories where OBS writes its recording segments and had Cobian copy those files into a separate vault folder on a BitLocker encrypted volume. The jobs use encrypted archives instead of plain file copies, which means that someone who walks off with the backup disk will still need the encryption password to read the contents.

Setting up retention inside Cobian turned out to be more delicate than I expected. Early in the process I misconfigured an option that attempted to delete files that were still in use, which obviously does not work well with long running recordings. After some trial and error, I arrived at a pattern where Cobian only touches closed recording segments and keeps a limited number of recent backup sets instead of trying to be a permanent archive. That approach fits the threat model. For a home system, it is more important to have a few days of reliable, redundant footage than a fragile attempt at a forever archive that will eventually fail.

From a design perspective, Cobian sits one step downstream from OBS. The recorder focuses on capturing and segmenting the video. Cobian focuses on taking those segments, encrypting them, and storing them on a different logical volume so that a single disk failure is less likely to wipe out everything.

5.5 WireGuard VPN and Remote Access

Remote viewing is deliberately constrained in this system. I did not want to forward RTSP or HTTP ports through the firewall or use a vendor relay service. Instead, I wanted a model where any remote device first joins my home network as a trusted peer and then accesses the recorder or other resources using the same paths as an internal device. WireGuard on the UniFi Dream Router is how I implemented that model.

On the router, I created a WireGuard server configuration and defined peers for my phone and laptop. Each peer has its own key pair and allows for IP configuration. When I bring up the VPN on my phone, it receives a virtual address inside a range that the UDR routes back into my main LAN. From the perspective of the Alienware recorder, a VPN connected phone looks like any other internal client. I can point a media player or browser at the recorder while I am away from home, and all of that traffic travels inside a WireGuard tunnel rather than through a series of ad hoc port forwards.

This arrangement fits the project's privacy goals. There is no externally facing web interface for the recorder or camera that an attacker can poke at. There is only a VPN endpoint on the router, which is specifically designed to handle that role. If WireGuard is not running, there is no remote access. If the VPN key on a device is compromised, I can revoke that peer from the UniFi interface without touching the recorder.

5.6 Logging and Security Controls

Logging and basic security controls are the quiet parts of the system that make everything else trustworthy. The UniFi Dream Router provides detailed firewall logs that show every blocked connection attempt from the camera network to the Internet. Those logs are one of

the main ways I verify that the segmentation and firewall rules are behaving as intended. When the network log analysis showed almost a thousand blocked connections from the Reolink camera within a short time window, it confirmed that my paranoia about phone home behavior was justified.

On the recorder itself, Windows event logs, OBS logs, and Cobian logs tell me whether recording and backup jobs are running reliably. When OBS froze during earlier tests, its log file contained clear hints about timeouts and stalled RTSP reads, which helped me track down the root cause. Cobian's logs show when backup jobs start, how long they take, and whether any files were skipped or locked.

Account and access controls tie this all together. The Alienware PC runs with a minimal set of local accounts, and the Cobian service uses a restricted identity. The UniFi controller and WireGuard configuration are protected behind strong credentials and, in practice, are only administered from inside the house. Combined with the network segmentation and the lack of exposed camera or NVR portals, these controls reduce the attack surface significantly compared to a typical plug and play camera deployment.

6. Implementation Process and Timeline

The design choices described in the previous sections did not appear all at once. They emerged over the course of the semester as I moved from a simple proof of concept to a system that could run for days at a time without manual intervention. The implementation process followed the same rough phases that I outlined in my project proposal and mid semester checkpoint, but with plenty of detours as real world behavior forced changes.

The first phase focused on getting a basic prototype running. I started with the simplest possible path from camera to disk. The Reolink E1 Pro was connected to my home network, and OBS on the Alienware PC was configured to subscribe to the camera's RTSP stream and write recording files to a local directory. At this stage I was not worried about segmentation, backups, or VPNs. I just wanted to see continuous video appear in a folder and confirm that the files were playable. Short test recordings quickly turned into longer sessions, and I iterated on OBS's encoding and container settings until I settled on a combination that produced reasonable quality and manageable file sizes.

Once the basic recording pipeline worked, I turned to network isolation. I created a dedicated surveillance network on the UniFi Dream Router, placed the camera on that VLAN, and adjusted firewall rules so that the new network could not reach the Internet. The Alienware recorder remained on the main LAN but was granted explicit permission to talk to the camera network on the necessary ports. This step took the camera out of the general

home network and put it into the constrained environment that the threat model called for. From that point forward, any time I wanted to adjust routing or access, I did so through UniFi, where I could see the effects in logs and interface views.

With the camera isolated and the recorder path stable for short runs, I began to stretch the system toward the continuous operation that a real NVR requires. OBS was left running for extended periods while I watched for freezes, growing log files, or other signs of trouble. It was during this phase that the long running freeze issue surfaced. The system might appear fine for hours and then quietly stop updating. Addressing that problem required a focused round of troubleshooting. I examined OBS logs, experimented with different source types, and eventually changed the RTSP transport to TCP with explicit reconnect and timeout parameters. Those changes were folded back into the main configuration and tested again in longer sessions until I was satisfied that freezes were an exception rather than the rule.

In parallel with stabilizing OBS, I implemented the backup layer. Cobian Reflector was installed on the Alienware PC and configured to run as a service so that it would continue to work even if no one was logged in. I created backup tasks that targeted the directories where OBS stores its recording segments and pointed them at a vault folder on a BitLocker encrypted volume. The first versions of these tasks were rough. Some options caused Cobian to interfere with files that OBS was still writing, which was not acceptable for a system that should be capturing events in real time. Through trial, error, and careful reading of the documentation, I refined the tasks so that Cobian only touched closed files and maintained a modest number of recent backup sets rather than trying to keep everything forever.

Remote access came next. Once the local recording and backup pipeline felt solid, I enabled WireGuard on the UniFi Dream Router and created peer configurations for my phone and laptop. The goal was to reach the recorder as if I were on my home LAN, without exposing any additional services directly to the Internet. After importing the WireGuard configuration on my phone, I tested the connection from outside the apartment. When the VPN was off, attempts to reach the recorder failed, which is exactly what I wanted. Once the VPN was on, the recorder responded as though I were on the local network. That pattern gave me confidence that remote access depended entirely on the VPN and not on any accidental port forwarding or relay services.

The last phase of implementation focused on operational polish. I configured OBS and Cobian to start automatically when the system boots so that a power interruption or planned reboot would not leave the system idle. I verified that the services come up in a reasonable order, that OBS reconnects to the camera, and that Cobian resumes its backup schedule. I also captured and archived the final versions of critical configurations, such as

firewall rules and WireGuard profiles, so that I could rebuild the system more quickly if I ever had to replace hardware.

By the end of this process, the implementation no longer felt like a set of fragile experiments. It resembled a small but coherent system with predictable behavior. Each phase built on the previous ones and forced me to revisit assumptions when reality disagreed with my diagrams. That iterative rhythm is as much a part of the project as any individual setting in OBS or Cobian. I treated each stable configuration as a small internal release, saving snapshots of key firewall rules, OBS profiles, and backup jobs so that I could roll back if a later change introduced new problems.

7. Testing, Troubleshooting, and Results

From the outside, this system looks simple: a camera, a recorder, a router, and a VPN. In practice, getting all of those parts to behave reliably over days instead of minutes required a fair amount of testing and troubleshooting. I did not run a formal test lab with synthetic traffic generators or multiple clients, but I did design a set of practical tests that matched the way I expect to use the system. Those tests focused on three questions: could the recorder capture a continuous stream for long periods without freezing, was the camera truly confined to a local only posture with all phone home behavior blocked, and could I safely reach the system over VPN from outside my apartment without accidentally exposing anything to the public Internet.

7.1 Test Strategy

The test strategy followed the project phases. Early on, as soon as I had the Reolink camera talking to OBS, I ran short functional tests. I would start a recording, wait a few minutes, stop it, and verify that OBS produced a playable file in the right folder. Once the basic path worked, I lengthened the sessions. Recordings ran for hours at a time while I went about my day. I would periodically check the preview window and the file system to see whether OBS was still writing segments and whether the resulting files played cleanly.

As the network design matured, I added isolation and security checks. After putting the camera on its own VLAN and adding outbound deny rules, I captured firewall logs from the UniFi controller and looked for any successful connections from the camera network to the Internet. I expected to see a large number of blocked attempts, and I wanted to see zero successful outbound sessions. When I turned on WireGuard and began using the VPN, I verified that I could no longer reach the recorder from outside the home network unless the VPN was active, and that once connected my phone could reach the recorder as if it were on the local LAN.

I also built in a few failure scenarios. I intentionally rebooted the router and the recorder while a recording was underway to see what would happen. After a router reboot, I watched to see whether OBS eventually resumed receiving frames from the camera or whether it stayed frozen on an old image. After a recorder reboot, I checked whether OBS and Cobian restarted correctly and whether the system resumed recording without manual intervention. These tests were not exhaustive, but they were enough to reveal where the configuration still had unacceptable failure modes.

7.2 OBS Stability and Freeze Analysis

The most serious technical issue that emerged during testing was OBS freezing during long running recordings. On short tests, the setup looked fine: The preview updated, files appeared on disk, and nothing seemed out of the ordinary. The problems only surfaced during extended sessions. I would leave the system recording and come back hours later to find that the preview was stuck on a single frame, and the interface was unresponsive. The recording indicator might still be lit, but in reality, the process had stopped writing useful data some time earlier.

Digging into the OBS logs and the underlying RTSP configuration made the pattern clearer. By default, the RTSP path from the Reolink camera to OBS was using UDP, and there were no explicit reconnect or timeout settings. When the camera or the network experienced a transient problem, the RTSP stream would stall. From OBS's perspective, the input thread was simply blocking, waiting for packets that never arrived. That stall propagated forward. The encoder waited for frames it was not receiving, the user interface tried to reflect a stream that had effectively died, and the whole application appeared frozen even though nothing technically crashed.

I treated this as a multi-layer problem rather than blaming OBS alone. The fix involved three main changes. First, I forced the RTSP transport to use TCP instead of UDP. TCP incurs a little overhead but is much better at smoothing over small network interruptions. Second, I added reconnect and timeout options so that if the stream did drop, the input would eventually give up waiting and try again rather than sitting in a blocked state indefinitely. Third, I adjusted the recording destination to favor a fast local volume and a simple directory structure, which reduced the chance that disk contention from backups or other processes would create additional pressure.

After these changes, I repeated the long running tests. I let OBS record for days at a time and monitored both the preview and the files on disk. When I scanned the logs, I could still see occasional connection hiccups, but they were accompanied by reconnect messages instead of an abrupt end of activity. The system was not perfect in the sense of being

invulnerable to every possible glitch, but in normal use it no longer froze silently and stopped doing its job.

7.3 Network and VPN Verification

While I was taming OBS, I was also verifying that the network behaved the way the design required. The central claim of the project is that this is a local only surveillance system that does not quietly leak data to vendor infrastructure. The only honest way to back that up is to look at the logs.

Once the camera was moved to its own VLAN and the outbound firewall rules were in place, I enabled logging for that policy on the UniFi Dream Router and let the system run. When I later exported and reviewed the logs, the results were striking. The Reolink E1 Pro attempted to open connections to hundreds of distinct external IP addresses in a relatively short timeframe. Every one of those attempts was marked as blocked by the firewall. That pattern is exactly what I wanted to see. It confirmed that the camera is indeed very chatty by default and that without explicit controls it would be talking to a large number of external services. It also confirmed that my rules were doing the work I expected them to do.

For VPN verification, I treated my phone as an ordinary remote client. With WireGuard disabled, I could not reach the recorder from the outside world at all. There were no port forwards or public web interfaces for me to hit. Once I enabled WireGuard and imported the peer configuration on my phone, the picture changed. I could bring up the VPN, receive an address inside my home network, and then reach the Alienware recorder as if I were sitting on my couch. When I dropped the VPN, access disappeared. That behavior gave me confidence that the only remote path into the system goes through the VPN endpoint on the router and that I am not accidentally exposing secondary services.

7.4 Storage and Retention Validation

Testing storage and retention is less dramatic than chasing down freezes, but it is just as important for a surveillance system. The first step was simply to ensure that OBS was writing files where I expected. With the final configuration in place, recordings are saved into dated directories on the hard disk, each file representing a fixed segment of time. To validate that the system could retain at least a full day of footage, I let it run for longer than that, then walked back through the directory structure and confirmed that recordings existed for all of the expected time ranges.

To get a sense of how far beyond twenty-four hours I could go, I combined those observations with the retention calculations I had done earlier based on bitrate and disk

size. The numbers on my slide suggested a theoretical window of roughly 12-14 days before the disk would be full if nothing were deleted or moved. That estimate is intentionally conservative. In reality, Cobian's backup jobs and any manual cleanup will influence how much data remains on the primary recording volume at any given time.

Cobian's behavior needed its own checks. After I corrected the early misconfiguration where it tried to touch in use files, I monitored backup runs to ensure that they completed without errors and that the resulting archives were both present and decryptable. I restored a small sample of files from the backup vault back into a test directory and played them to confirm that they were not corrupted. Between OBS's primary recording directory and the encrypted backups, I ended up with two independent locations for recent footage, which is an acceptable level of redundancy for a single camera home system of this scale.

7.5 Goals and Outcomes

At the proposal stage, I set a handful of goals that defined success for this project. By the end of the semester, I had a clear sense of which ones were fully achieved, which ones were partially achieved, and which ones changed as I learned more.

The clearest success is the local only posture. The final system records all video on a machine in my apartment, backs it up to a BitLocker protected volume, and does not send footage to any vendor cloud. The firewall logs show that the camera's attempts to reach external servers are blocked. Remote viewing only works after a WireGuard tunnel is established, and there are no exposed ports for a casual attacker to discover.

Continuous recording is another area where the system meets its original goals. With the OBS stability fixes in place, the recorder can run for days without freezing, and it retains at least a full day of footage with practical headroom into a multi-day window. The exact number of days depends on the chosen bitrate and how aggressively I manage storage, but the core requirement of being able to go back and review at least the last twenty-four hours has been satisfied.

Some ideas remained more aspirational. Early on I imagined enforcing a very strict limit on how much of the archive could be viewed remotely, such as only allowing a few minutes of history when connected over VPN. That concept influenced how I thought about privacy, but I did not implement a hard technical control that cuts off playback at a fixed offset. In practice, the VPN based model and my control over client devices reduce the risk of casual browsing, but the system would need additional logic to enforce the original five-minute limit exactly.

There are also goals related to automation and monitoring that are only partially realized. OBS and Cobian start automatically with the system, and they are configured in a way that makes long running operation realistic. What I do not yet have is a full health monitoring stack that alerts me if recordings stop or backups fail. For the scope of this course, that level of automation was out of reach, but the design leaves room to add it later.

Overall, the testing and troubleshooting process took the project from an idea that sounded good on paper to a working system that behaves predictably under normal conditions. It exposed weaknesses in my initial configuration, especially around long running RTSP capture and backup interference, and it provided hard evidence that the camera behaves exactly like a privacy conscious person would fear when left unchecked. The observations, log snippets, and configuration changes described in this section function as my informal testing and bug report history for the system. The final result is not perfect, but it does meet the core requirements I set for myself, and it provides a solid foundation for future improvements.

8. Risk Analysis

Designing a privacy first surveillance system is partly about adding controls and partly about being honest about what can still go wrong. Moving away from a vendor cloud removes one class of risks, but it also shifts responsibility onto me as the system designer and operator. In this section I walk through the main risk categories that apply to the system in its current form, how the design mitigates them, and what remains as residual risk.

8.1 Privacy and Legal Risk

The entire project is framed by a desire to reduce privacy risk compared to a typical commercial camera deployment. By keeping all recordings on a local machine and blocking the camera from contacting external servers, I avoid the specific dangers of vendor employees or attackers browsing a cloud archive of my home. The UniFi firewall logs make it clear that the Reolink camera will attempt to phone home to a large number of IP addresses when given the chance, and that those attempts are being denied on my network. That is a concrete reduction in exposure compared to a default configuration.

At the same time, local storage creates its own privacy and legal questions. If someone gains access to the recorder, physically or through a compromise, they gain a very detailed record of life inside the apartment. BitLocker and Cobian’s encrypted backups make that harder, but they do not make it impossible. There is also the legal reality that if a serious incident occurs and law enforcement believes the footage is relevant, the disks themselves

could be seized. In that scenario, the encryption keys become the main line of defense. I made a conscious choice to use strong disk and backup encryption so that anyone who wants to review the footage needs both the hardware and the keys, but that still leaves open questions about compelled access that goes beyond the scope of a technical project.

Another privacy risk comes from misconfiguration. The design assumes that the camera network has no route to the Internet and that the recorder has no exposed services on the public side of the router. If I later add new rules, change VLANs, or plug in additional services without the same level of care, it is easy to accidentally create a path that did not exist before. The system depends on me continuing to respect the original threat model as I modify my home network over time.

8.2 Technical and Operational Risk

On the technical side, the biggest structural risk is that the system revolves around a single recording host. The Alienware PC does all of the work. If its motherboard fails, if its power supply dies, or if the machine is accidentally powered down for a long period, the camera will keep running, but nothing will be capturing the stream. There is no redundant NVR standing by to take over. For a one camera home system that may be an acceptable tradeoff, but it is still a single point of failure.

Individual components inside that host also matter. A failing hard drive can silently corrupt recordings or backups. An operating system update can break drivers or change network behavior in ways that cause OBS or Cobian to misbehave. During the project I already saw how small network glitches could cause OBS to freeze when the RTSP path was not configured correctly. The final configuration is much more robust, but it is not immune to all possible failures. In practice, the system relies on me noticing when something is wrong, reading logs, and adjusting settings to restore stability.

Operationally, there is a risk with having one person as the only administrator. The configuration lives in my head and in scattered notes and exports. If I am unavailable, there is no one else who can quickly rebuild or troubleshoot the system. There is also the risk of simple neglect. If I disable logging to save space, ignore backup failures, or forget to test VPN access after router changes, the system can drift away from its intended security posture without an obvious alarm.

8.3 Financial Constraints and Tradeoffs

Budget constraints were part of the design from the very beginning. I reused an old gaming PC, selected a reasonably priced camera, and chose a router that was more expensive

than a typical ISP box but still within reach for a graduate student who cares about networking. Those decisions kept out of pocket cost manageable, but they also closed off some options.

With a larger budget, I could have purchased a dedicated NVR appliance with built-in health monitoring, redundant disks, and vendor support, or enterprise grade cameras with more transparent firmware and hardening options. I could also have invested in a UPS sized specifically for the recorder and router to ride through power outages cleanly. Instead, I chose to spend my time making consumer grade hardware behave like a more professional stack. That tradeoff aligns with the educational goals of the project, but it does mean that some reliability and maintainability features that money can buy are instead being covered by my own attention and effort.

8.4 Residual Risk and Future Mitigations

Even with careful design, logging, and encryption, some risks remain. A determined attacker with physical access and enough time can still attempt to extract data from encrypted disks. A major change in my ISP equipment or home network layout could introduce new paths around the router if I am not deliberate about recreating the same segmentation. Long term component aging will eventually demand hardware replacements, and each replacement is an opportunity to get a detail wrong.

The key mitigation for these residual risks is ongoing discipline. That means periodically reviewing firewall rules and logs to confirm that the camera is still confined to its own network, verifying that backups are not only created but restorable, and treating major changes to the network or hardware as small projects rather than casual tweaks. Over time I would also like to reduce some of the single host risk by adding better monitoring and, eventually, more structured documentation so that this system does not depend entirely on my memory.

The important point is that moving away from cloud cameras does not magically make risk disappear. Instead, it trades one set of risks for another. In this project, I intentionally chose risks that I can see and influence. The local only design, strong encryption, and segmented network give me tools to manage those risks directly, rather than hoping that a vendor I have never met will manage them on my behalf.

9. Design Rejects and Alternatives Considered

I did not arrive at the final design by picking the first camera, router, or software stack that looked promising on paper. A significant part of the project involved exploring alternatives

and then deciding if they were not a good fit for the privacy, cost, or reliability targets I had set. This section describes the main options I considered and rejected, and why they ultimately stayed out of the final system.

9.1 Cloud Platforms and Managed Ecosystems

The most obvious alternative to building my own system was simply buying into a mainstream cloud camera ecosystem. A couple of commercial platforms would have given me sleek cameras, polished mobile apps, and turnkey remote access with minimal configuration. The problem is that those conveniences are inseparable from the things that concern me most. To function, those platforms need to send live or near live video into vendor infrastructure. Authentication and authorization are handled by the vendor, not by me. Employees and contractors have access to internal support tools that let them look into customer environments, and history has shown that those tools can be abused.

From a purely user experience perspective, a cloud platform would have solved my partner's problem in a weekend. From a privacy perspective, it would have created a long-term dependence on systems I can neither inspect nor control. The network logs from my own camera experiments reinforced that worry. If a relatively inexpensive Reolink camera, running in a local only configuration, tries to contact hundreds of external IP addresses when left to its own devices, it is difficult to imagine that a fully cloud oriented camera is behaving in a more conservative way. That is why none of the big commercial ecosystems made it past the initial brainstorming stage for this project.

9.2 Software NVRs and All in One DVR Solutions

On the software side, I also looked at purpose-built NVR packages that promise to turn a generic machine into a full featured DVR. Projects like Agent DVR and Shinobi were on my radar at various points. They typically offer a web-based configuration interface, multi-camera support, motion detection, and all the features we associate with small business surveillance systems.

There were two reasons I did not adopt one of these tools as the core of my project. The first was complexity. Many of these platforms assume a user who is committed to running a dedicated NVR box or container stack and who is comfortable living inside that ecosystem long term. They often bring their own database, their own recording pipeline, and their own upgrade and backup story. That can be powerful, but it also means inheriting a lot of moving parts that are not necessary for a single camera home system. The second reason was alignment with my privacy model. I wanted a simple, inspectable pipeline from RTSP stream to recording file, plus a separate backup step that I could reason about independently. OBS and Cobian, combined with the operating system's own encryption

and access controls, gave me that transparency. A more monolithic NVR, while technically capable, felt like the wrong fit for my goal of understanding and owning every major piece of the stack.

9.3 Routers and Network Appliances

Routers were another area where I tried and evaluated several options before deciding on the UniFi Dream Router. My ISP provided a basic gateway that could have been pressed into service, and I experimented with consumer grade routers that advertised strong Wi-Fi and reasonable security features. While those devices could route traffic and provide wireless connectivity, they tended to fall short when I tried to implement the specific network segmentation and logging the project required.

Creating a dedicated camera VLAN, applying granular firewall rules, and then actually inspecting per device logs to see what the camera was doing all at once is not a common use case for a budget router. In some cases, the user interface resisted that level of detail, in others the logging was too coarse to be useful. Documentation for advanced VPN features was often thin or oriented toward older protocols that I was not interested in deploying at home. The UniFi Dream Router, by contrast, treated VLANs, firewall policies, and WireGuard as first class citizens. It still required configuration and testing, but it did not fight me at the basic level. That is why the other routers ended up in my personal “rejects pile” while the UDR became the backbone of the final design.

9.4 Secure Streaming Options

Before I committed to a VPN centric model for remote access, I considered using application level secure streaming as a way to reach the camera and recorder. At various points I looked at tools like Jami and at the idea of having OBS stream to external platforms or relay services. These options promised encrypted transport and, in some cases, peer-to-peer connectivity without a traditional VPN.

The more I thought through the details, the less appealing they became. Application-level streaming introduces another layer of configuration and potential misconfiguration. If I build a pipeline where OBS or some other client streams to a third-party service, I am back in the situation I was trying to avoid: remote infrastructure is in the path of video leaving my apartment. Peer to peer tools that rely on their own rendezvous servers or overlay networks also carry their own privacy and reliability tradeoffs, and they generally assume an ecosystem that extends beyond my own router and recorder.

In the end, WireGuard gave me a cleaner model. When I am remote, I join my home network through a VPN that I operate, then I use ordinary tools to reach the recorder. There

are no extra web portals or streaming accounts to manage, and there is no relaying through cloud services that I cannot fully audit. The secure streaming options I explored helped clarify what I did not want, which made the VPN based design feel more solid once I implemented it.

9.5 Tradeoff Summary

Looking back, the common thread among the rejected options is that they either outsourced trust to vendors or added more complexity than I needed for a single camera system. Cloud platforms would have taken privacy out of my hands entirely. All in one NVR software would have buried the recording and backup logic under layers of abstraction. Simpler routers and application-level streaming tools would have made it harder to enforce and verify the network boundaries that anchor this project. The final design is not the easiest to set up, but it is the one that matches my comfort level with who can see my data and how the system behaves when something goes wrong.

10. Struggles and Lessons Learned

No meaningful project is just a straight line from proposal to polished demo. This one certainly was not. Along the way I ran into technical problems that did not have obvious fixes, selection decisions that I second guessed repeatedly, cost constraints that forced compromises, and plenty of moments where the easiest solution would have been to give up and buy a commercial kit. This section captures the most important struggles and what they taught me.

10.1 Technical Struggles

The most visible technical struggle was getting OBS to behave like a reliable recorder instead of a temperamental streamer. On paper, pointing OBS at an RTSP source and pressing record is trivial. In practice, long running sessions exposed edge cases that short tests never hit. The freezing issue described in the testing section was not just an annoyance. It undermined the basic promise of the system. If my recorder cannot be trusted to stay awake while I am gone, it does not matter how elegant the rest of the design is.

Tracking down the root cause forced me to dig into OBS logs, read about RTSP transport modes, and think more carefully about how network hiccups propagate into application behavior. The fix turned out to be a combination of relatively small changes: forcing RTSP over TCP, setting reasonable timeouts and reconnect behavior, and adjusting recording destinations. None of those changes would have meant much on their own. Together they

made the difference between a system that quietly failed and one that could ride through minor disturbances. That experience reinforced the idea that reliability often comes from layering several modest improvements rather than relying on a single magic setting.

Storage and backup were another area where the initial plan looked better on paper than in practice. I initially tried backup configurations that interfered with files OBS was still writing. That created errors, partial backups, and in some cases confusion about which copies were actually safe to delete. Untangling that mess required a clearer mental model of how OBS segments recordings, how Cobian decides which files to touch, and what it means for a file to be “closed” in a long running recording environment. It reminded me that backup tools are sharp instruments. Used carelessly, they can damage the very data they are meant to protect.

10.2 Selection Struggles

The selection problems were less dramatic technically, but just as frustrating. Choosing a camera, router, and software stack in a crowded market is not easy when your priorities are different from the target customer’s priorities. Most marketing material for cameras focuses on picture quality, app features, and cloud integration. Very little documentation talks frankly about network behavior, local only use, or long-term support. That meant I had to rely on a mixture of vendor documentation, user forums, and my own testing to decide what to trust.

The routers were a similar story. The default ISP gateway and several consumer routers looked fine on the surface. They could handle normal home tasks effectively. Only when I tried to set up VLANs, apply detailed firewall policies, and enable modern VPNs did their limitations become obvious. I spent more time than I expected reading manuals, browsing community posts, and experimenting with half documented config options before accepting that I needed a device built with that kind of use case in mind. The UniFi Dream Router was not the first router I looked at, but it was the first one that seemed genuinely comfortable with what I wanted it to do.

These selection struggles taught me that there is a real gap between what most consumers are sold, and what someone with a strong privacy and security bias actually needs. It also reinforced the value of logs as a decision-making tool. Seeing the Reolink camera’s outbound connection attempts in the UniFi logs did more to validate my concerns than any spec sheet could.

10.3 Cost and Constraint Struggles

Cost was a constant background presence throughout the project. I did not have the budget to buy enterprise grade cameras, a dedicated NVR chassis, or a rack full of redundant storage. That meant working with an older PC, a modest camera, and a single router that had to serve both as my home gateway and as the core of this experiment.

These constraints forced trade-offs. I could not simply throw hardware at problems. When OBS struggled, the solution had to come from configuration, not from upgrading to a more powerful machine. When the storage space felt tight, I had to adjust recording parameters and backup strategies instead of buying additional disks on impulse. In some ways those constraints made the project harder. In other ways, they made it more educational. It is one thing to design a system with ideal hardware, and another to make something robust out of components that many people would consider obsolete.

The main downside of this approach is that some desirable features remain out of reach for now. I do not have a redundant recorder. I do not have a specialized UPS sized perfectly for this stack. I do not have professional support contracts. What I have instead is a system whose behavior I understand at a level that would not be necessary if I simply outsourced the problem.

10.4 Lessons Learned

Several lessons emerged through these struggles that will stay with me beyond this project. The first is that network logs are not optional when you care about privacy. Without the UniFi firewall logs, I would have no concrete understanding of how aggressively the camera tries to contact external servers. With them, I have direct evidence that the local only firewall policy is actually doing something. The difference between “I think this is safe” and “here is a log showing that unwanted traffic is being blocked” is significant.

The second lesson is that the old security mantra of “trust but verify” does not translate well into the privacy domain. With a cloud platform, you often cannot verify much of anything. You see the user facing app, but not the internal tooling or the data flows behind it. For me, the safer assumption is that systems will do more than they advertise, and that it is my job to constrain them. Treating the camera as untrusted from the outset led directly to the isolated VLAN and strict firewall rules that define this project.

Another lesson is that low budgets cost time. The hours I spent coaxing an older PC and consumer hardware into acting like a privacy aware NVR would have bought a polished commercial system many times over. In a classroom and learning context, that trade is

acceptable and even beneficial. In a production environment, the calculus might be different. It is useful to understand both sides of that equation.

Finally, this project sharpened my sense of how far hands on tinkering can take you and where more structured research is needed. Trial and error are useful up to a point, but stubborn problems like the OBS freezes only yielded when I combined experimentation with careful reading of documentation and logs. The balance between exploration and deliberate study is something I will carry forward into future work.

11. Conclusion and Future Enhancements

At the proposal stage, this project was an idea about building a safer alternative to cloud cameras. By the end of the semester, it had become a working system that actually runs in my apartment. The Reolink E1 Pro feeds an RTSP stream into OBS Studio on a repurposed Alienware desktop. That desktop records continuous video in segmented files, backs them up to an encrypted volume via Cobian Reflector, and is reachable remotely only through WireGuard VPN sessions terminated on a UniFi Dream Router. The router segments the camera onto its own network and blocks all of its attempts to talk to the wider Internet. Logs and tests demonstrate that this configuration behaves as intended under normal conditions.

From a requirements perspective, the system meets the core goals I set early on. It keeps footage local, under my control, rather than in a vendor cloud. It retains at least a full day of history with practical headroom into a multi-day window, which gives me time to notice and respond to events. It lets me view a live feed remotely, but only after I have joined my home network through a VPN that I operate. It does not expose camera or recorder portals directly to the public Internet. These properties distinguish it sharply from the commercial kits that initially raised my concerns.

The work has also been valuable from a professional standpoint. It pulled together topics that usually live in separate corners of a curriculum. I had to think about network design and firewall rules, long running application behavior, operating system hardening, backup and retention policies, and the human factors that come with deploying a system in a real home. It turned abstract discussions about privacy and control into concrete configuration decisions that I can see reflected in logs and real traffic.

At the same time, it is clear that there is room for improvement. The current design uses a single recorder and a single camera. Adding additional cameras would raise new questions about bandwidth, storage, and management. A more robust system might include a second recording host or at least a cleaner path for quickly standing up a replacement if

the Alienware machine dies. Power protection is another obvious area to extend. A properly sized UPS for the router and recorder would smooth out the kinds of interruptions that originally triggered OBS freezes and reduce the risk of abrupt shutdowns during backups.

Monitoring and alerting are also missing pieces. Right now, the system is quiet when it is healthy and equally quiet when something fails. A future enhancement would be to add lightweight health checks and notifications so that I receive a message if recording stops, if backups fail repeatedly, or if the camera suddenly starts generating unusual traffic. That kind of visibility would help close the gap between a classroom project and an operational system that someone could rely on for years.

Finally, there is the opportunity to package what I have learned into something that others can use. Documenting the configuration more formally, generalizing the design to support multiple cameras, and publishing a guide for building a similar local only system with common hardware would all be natural extensions. Doing so would make the project useful beyond my own apartment and contribute, in a small way, to a wider conversation about what private surveillance should look like in practice.

This written report is paired with an accompanying slide deck that I used for the course presentation. The deck, titled “Secure Home Surveillance System Project,” contains the screenshots, diagrams, and configuration views that illustrate the system described here, including UniFi firewall views, OBS and Cobian settings, and views of the running camera stream. Together, the report and the slide deck form a complete record of the project, with the report providing the narrative and the slides providing the visual context. You can view it in Microsoft Word below or at the following link: MSIS Project.pptx