RADIUS/UDP Considered Harmful

Sharon Goldberg
Cloudflare

Miro Haller
UC San Diego

Nadia Heninger
UC San Diego

Mike Milano
BastionZero

Dan Shumow
Microsoft Research

Marc Stevens
Centrum Wiskunde & Informatica

Adam Suhl
UC San Diego

Abstract

The RADIUS protocol is the de facto standard lightweight protocol for authentication, authorization, and accounting (AAA) for networked devices. It is used to support remote access for diverse use cases including network routers, industrial control systems, VPNs, enterprise Wi-Fi including the Eduroam network, Linux Pluggable Authentication Modules, and mobile roaming and Wi-Fi offload.

We have discovered a protocol vulnerability in RADIUS that has been present for decades. Our attack allows a man-in-the-middle attacker to authenticate itself to a device using RADIUS for user authentication, or to assign itself arbitrary network privileges. Our attack exploits an MD5 chosen-prefix collision on the ad hoc RADIUS packet authentication construction to produce Access-Accept and Access-Reject packets with identical Response Authenticators, allowing our attacker to transform a reject into an accept without knowledge of the shared secret between RADIUS client and server.

We optimize the MD5 chosen-prefix attack to produce collisions online in less than five minutes, and show how to fit the collision blocks within RADIUS attributes that will be echoed back from the server. We demonstrate our attack in a variety of settings against popular RADIUS implementations.

It is our hope that this attack will provide the impetus for vendors and the IETF to deprecate RADIUS over UDP, and to require RADIUS to run over secure channels with modern cryptographic privacy and integrity guarantees.

1 Introduction

The RADIUS (Remote Authentication Dial-In User Service) protocol is at the core of today's network infrastructure. Although the protocol was first designed in 1991—during the era of dial-up internet—it remains the de facto standard lightweight authentication protocol used for remote access for users and administrators to networked devices. RADIUS is supported by "essentially every switch, router, access point, and VPN concentrator product sold in the last twenty years" [19].

RADIUS is a critical part of modern telecommunications and enterprise networks. In large enterprises, RADIUS may control access to tens of thousands of switches. Essentially all ISPs offering DSL or FTTH (Fiber to the Home) use RADIUS, as do 802.1X and Wi-Fi authentication [16, 51]. RADIUS is used for 2G and 3G cellular roaming [4, 9] and 5G DNN (Data Network Name) authentication [51]. (3G+ networks may use its successor Diameter.) Mobile Wi-Fi offload with SIM card-based authentication uses RADIUS [16, 51], as does private APN (Access Point Name) authentication, which is used by IoT devices, companies, fire and rescue services, or law enforcement to gain access to non-public networks [37, 45]. SITA uses RADIUS to authenticate passenger aircraft worldwide for telemetry data [44]. Siemens and GE smart grid products use RADIUS [1, 25]. Eduroam shares Wi-Fi to roaming users at educational institutions in 100 countries through a global hierarchy of networked RADIUS servers that saw 7.5 billion authentications in 2023 [32]. The Wireless Broadband Alliance OpenRoaming federation also uses RADIUS.

Much of this RADIUS traffic is sent over UDP. The first hop between a NAS (Network Access Server) and RADIUS server is "almost always" over UDP [16]. While it is possible to send RADIUS traffic over TLS or IPsec, this is typically only done for packets traveling over the open internet; currently only rare security-oriented organizations use RADIUS/TLS internally [16, 51]. Although sending RADIUS/UDP traffic over the internet is discouraged, it still happens. "A majority" of country eduroam networks still send RADIUS/UDP traffic on the internet [38] as do some "cloud" providers [18]. At least one major RADIUS vendor has no TLS support at all [16].

1.1 Our Attack

The core of the RADIUS protocol predates modern secure cryptographic design. Surprisingly, in the two decades since Wang et al. [52] demonstrated an MD5 hash collision in 2004, RADIUS has not been updated to remove MD5. In fact, RADIUS appears to have received notably little security analysis given its ubiquity in modern networks.

In a RADEXT working group Internet-Draft from 2023 entitled "Deprecating Insecure Practices in RADIUS" [18] Alan DeKok, the lead maintainer of FreeRADIUS—the most widely used RADIUS implementation—wrote

> As of the writing of this specification, RADIUS/UDP is still widely used, even though it depends on MD5 and "ad hoc" constructions for security. While MD5 has been broken, it is a testament to the design of RADIUS that there have been (as yet) no attacks on RADIUS Authenticator signatures which are stronger than brute-force.

In this paper, we give an attack against the ad hoc RADIUS Response Authenticator "signature" (actually a MAC) construction. This protocol flaw has been present in RADIUS since the earliest versions [43].[1]

In RADIUS, a NAS acts as a client that verifies an end user's credentials via RADIUS requests to a central server. The server responds with an accept or reject message (or a further challenge). Our attack allows a man in the middle between the RADIUS client and server to forge a valid Access-Accept response to a failed authentication request. This forgery will cause the NAS to grant the adversary access to network devices and services without the adversary guessing or brute forcing passwords or shared secrets.

Our attack combines an MD5 chosen-prefix collision [48] with a protocol exploit that allows us to hide the collision gibberish in an attribute that is guaranteed to be echoed in the server response. The Response Authenticator used to integrity protect server responses is an MD5 hash of several values including a random client nonce, protocol attributes, and a fixed secret shared between client and server. Our man-in-the-middle attacker injects a maliciously constructed attribute into the request that is designed to produce an MD5 collision between the Response Authenticator authenticating the server's legitimate response and the attacker's forged response.

To carry out this attack, the attacker must compute an MD5 chosen-prefix attack online, before the client times out, because the hash collision includes a random session nonce.

We are able to compute this MD5 chosen-prefix attack in about five minutes by making several improvements to Stevens's [46] open-source `hashclash` chosen-prefix attack implementation. We give a novel method to force bytes in collision blocks that allows us to fit the resulting MD5 collisions into the bounded length of a RADIUS protocol attribute.

Although we have stopped our engineering efforts at the five minute mark, our reported running times are a generous upper bound on the capabilities of a well-resourced attacker. Every step of the collision algorithm parallelizes well. An attacker able to implement the attack on GPUs, FPGAs, or hardware would be tens to hundreds of times faster.

We carried out proof of concept attacks against illustrative use cases of RADIUS. These include the default FreeRADIUS client and server implementation, the Okta service's RADIUS integration, a Cisco ASA firewall, and Linux PAM authentication via RADIUS. These targets were chosen due to availability; essentially all implementations are vulnerable by default because the underlying vulnerability is in the protocol.

[1] Although the chosen-prefix MD5 collision attack we use dates from 2007, the first weakness in the MD5 hash function was publicly documented in 1993, SHA-1 was published in 1995, and HMAC dates from 1996 [6]. RFC 5080 [34] noted in 2007 that Access-Request packets not containing a Message-Authenticator attribute may be trivially forged.

1.2 RADIUS/UDP should be deprecated

The `hashclash` software that we build on was also used in 2009 and 2016 to demonstrate attacks against TLS certificates [48] and protocol transcript hash authentication for TLS and SSH [7]. The MD5 hash function has been known not to be collision resistant for two decades, and has had known weaknesses for three. Today, a lack of explicit attacks against the use of MD5 in a protocol is more likely to be a sign of lack of cryptanalytic attention than any robust notion of security.

DeKok's Internet Draft continues

> We recognize that RADIUS/UDP will still be in use for many years, and that new standards may require some modicum of privacy. As a result, it is a difficult choice to forbid the use of these constructs. If an attack is discovered which breaks RADIUS/UDP (e.g. by allowing attackers to forge Request Authenticators or Response Authenticators, or by allowing attackers to de-obfuscate User-Password), the solution would be to simply deprecate the use of RADIUS/UDP entirely. It would not be acceptable to design new cryptographic primitives in an attempt to "secure" RADIUS/UDP.

We give such an attack in this paper. It is time to deprecate RADIUS over UDP and replace it with a transport mechanism that provides modern cryptographic security.

1.3 Contributions

- We develop a novel protocol attack against RADIUS that allows arbitrary forgery of Response Authenticators. While MD5 has been known to not be collision resistant for two decades, it was not known how to exploit this property in the context of the RADIUS protocol.
- We optimize the MD5 chosen-prefix attack computation in order to carry out chosen-prefix attacks in minutes.
- We develop a novel technique for forcing bytes in MD5 collision blocks, which allows us to optimize our collision attack and fit the resulting collisions within the length restriction of RADIUS protocol attributes.
- We have tested our attack against common RADIUS client and server implementations and use cases.

1.4 Summary and Impact

Nearly all RADIUS/UDP implementations are vulnerable to our protocol attack when using non-EAP authentication methods; see Section 7.1. RADIUS Accounting and EAP authentication appear less practically exploitable, although a theoretical protocol vulnerability may exist. Our attack also requires man-in-the-middle network access. Not all of the example RADIUS deployments above in Section 1 are practically exploitable; organizations should independently verify.

1.5 Disclosure

We reported the vulnerability to the IETF and CERT on February 2, 2024. CERT is coordinating the disclosure process among affected vendors; as of this writing 86 potentially affected vendors have been identified and notified. We spoke to Alan DeKok of FreeRADIUS and the IETF RADEXT working group on February 5; he confirmed the vulnerability and we have been in dialog about impact and mitigations. DeKok has authored a white paper detailing practical mitigation considerations for implementers, vendors, and admins [19].

Patches implementing the Message-Authenticator mitigation described in Section 7.1 will be available from major RADIUS implementations in coordinated release with this work; we expect future versions of [18] to update mandated client and server behavior. The IETF RADEXT working group has existing drafts in progress outlining RADIUS/(D)TLS [38]. Ultimately, we hope our work will hasten the formal deprecation of RADIUS/UDP and the standardization of RADIUS/(D)TLS.

1.6 Ethical Considerations

The attack demonstrations we describe in this paper exploit clients that we set up ourselves to test. The server queries we make to remote servers we do not control (e.g., our demonstration against Okta) send only normal protocol messages for registered user accounts and receive legitimate responses; our forgery attack targets the clients processing these responses.

The `hashclash` chosen prefix collision implementation has been public since 2009. Our improvements to the chosen prefix MD5 collision are available at [46]; we do not plan to publish full end-to-end exploit scripts for RADIUS itself.

2 Background

This section provides background on the RADIUS protocol and reviews related work as context for our attack.

2.1 The RADIUS Protocol

The RADIUS protocol provides authentication, authorization, and accounting (AAA) services for end users and administrators on network access servers via a central RADIUS server.

We begin by summarizing the relevant parts of the RADIUS protocol described in RFC 2865 [42], which specifies the Password Authentication Protocol (PAP).

In a typical setting, an end user wishes to authenticate to a networked device by entering a username and password at a login prompt. The networked device is a trusted party in the RADIUS protocol, and is running RADIUS client software. It shares a fixed shared secret with the RADIUS server.

PAP Authentication. The RADIUS client constructs an Access-Request packet that includes a one-byte ID, a 16-byte random nonce called a Request Authenticator, and additional fields (called attributes) that may include the username, user-entered password (encoded with an ad hoc MD5-based obfuscation function), and connection request information such as Network Access Server (NAS) IP address, NAS port, or other request information for the server to process.

The client then sends the request in the clear over UDP to the server. As originally written in RFC 2865 [42], this packet has no encryption outside of the password obfuscation and no integrity check. The server will check whether the source IP address matches a known client; otherwise the request is entirely unauthenticated.

The RADIUS server then processes the Access-Request. This may include looking up the user's password in a database, or querying an authentication server. If the server rejects the request, the server responds with an Access-Reject message. If the server accepts the request, it responds with an Access-Accept message that may contain attributes specifying metadata or configuration options for the connection.

Response Authenticator. Both Access-Reject and Access-Accept packets include a value called a Response Authenticator which is designed to prevent the forgery of responses. The Response Authenticator is computed as $\mathrm{MD5}(\texttt{Code} \,\|\, \texttt{ID} \,\|\, \texttt{Len} \,\|\, \texttt{ReqAuth} \,\|\, \texttt{Attributes} \,\|\, \texttt{Secret})$ where the Code, ID, Length, and Attributes are copied directly from the Access-Accept or Access-Reject packet, the Request Authenticator `ReqAuth` is a 16-byte random nonce included in the corresponding Access-Request packet, and the Shared Secret `Secret` is the fixed shared secret known to client and server.

Message-Authenticator. RFC 2869, "RADIUS Extensions" [54] defines a Message-Authenticator attribute that contains an HMAC-MD5 tag computed over the entire packet using the shared secret as the key. However, this attribute is optional for RADIUS access requests and responses not using EAP and many implementations do not require it by default.

Proxy-State Attribute. Access-Request packets may also contain one or more Proxy-State attributes. RFC 2865 [42]

specifies in numerous locations that any Proxy-State attributes present in an Access-Request "MUST be copied unmodified and in order into the response packet." Each Proxy-State attribute consists of a one-byte code followed by a one-byte length field and a string of at most 253 arbitrary bytes.

EAP Authentication. Extensible Authentication Protocol (EAP) support in RADIUS is defined in RFC 2869 [54] (RADIUS Extensions) and RFC 3579 [8] (EAP & RADIUS) updated in RFC 5080 [34]. A number of EAP authentication methods exist and are commonly used, including EAP-TLS and PEAP. The details of different EAP authentication methods are complex, and are largely orthogonal to our attack.

When EAP is used for authentication, the Access-Request message contains an EAP-Message. For RADIUS/UDP this is sent in the clear via UDP. Some EAP authentication methods may use encryption to protect authentication credentials or user information, but the final EAP Success or Failure message is not encrypted or integrity checked. Once the EAP authentication has terminated, the RADIUS server sends an Access-Accept or Access-Reject packet via UDP which contains an EAP-Message containing EAP-Success or EAP-Failure. RFC 2869 [54] specifies that all packets containing an EAP-Message must also contain a Message-Authenticator attribute, or else they must be silently discarded.

Interestingly, RFC 3579 [8] contains a section on conflicting messages between RADIUS and EAP: "The NAS MUST make its access control decision based solely on the RADIUS Packet Type (Access-Accept/Access-Reject)." Examples include "If the NAS receives an Access-Accept with an encapsulated EAP Failure, it will grant access to the peer."

In a common setup, the EAP Success will be sent by the NAS to the end-user device, while the Access-Accept is interpreted by the NAS. Thus an Access-Accept with EAP Failure could result in the two devices having differing views on whether authentication succeeded or failed. However, in 802.1X, the EAP session also derives a master key that is shared from the RADIUS server to the NAS, and future communication is encrypted with that key. An attacker who bypasses EAP authentication to forge an Access-Accept with an EAP Success would lack this negotiated key and thus be unable to access the network.

RADIUS Accounting RADIUS Accounting, specified in RFC 2866 [39], allows a NAS to use a central server to log network service usage and traffic statistics. This accounting information can be used for billing or auditing.

A RADIUS Accounting client sends an Accounting-Request packet at the start and end of a user's session. Accounting-Request packets are structured similarly to Access-Request packets, except the Request Authenticator is $\mathrm{MD5}(\texttt{Code} \,\|\, \texttt{ID} \,\|\, \texttt{Len} \,\|\, 0^{128} \,\|\, \texttt{Attributes} \,\|\, \texttt{Secret})$ instead of a random nonce. This field is intended to prevent an attacker from forging an Accounting-Request packet with false accounting information: if the Request Authenticator is incorrect, the accounting server ignores the packet.

Response Authenticators from Accounting-Response packets are computed as in Section 2.1. Other types of accounting packets (e.g. CoA and Disconnect) are computed similarly to Accounting-Requests.

2.2 Security analysis of RADIUS

The RADIUS protocol has received surprisingly little academic attention given its widespread use.

Hill analyzed RADIUS in 2001 [26]. He surveyed numerous protocol issues including brute force attacks against the shared secret or password from an attacker who observes the MD5-obfuscated password [36], a precomputation-based brute force attack on Request Authenticators, that repeated Request Authenticators can compromise user passwords, and that short shared secrets are likely to be insecure.

DeKok enumerates these and other vulnerabilities [18]. These include information sent in cleartext, that MD5 is broken, that shared secrets can be brute forced, that Message-Authenticators can be inconsistently used and that Access-Request packets missing Message-Authenticators can be forged or replayed, and that proxies expose MD5-obfuscated user passwords as they de-encode and re-encode them.

We will describe the offline brute-force attack against the shared secret by an adversary who observes a request and corresponding response packet containing a Response Authenticator. Since all of the information used to compute the MD5 hash for the Response Authenticator is present in the request and response except for the shared secret, the attacker can simply brute force shared secret values. The cost to guess an n-character shared secret consisting of printable ASCII characters is $95^n \approx 2^{6.6n}$. For a 7-character secret, this is 2^{46}; for a 10-character secret this is 2^{66}.

Most of the cryptographic security analysis of RADIUS has focused on the MD5-based password obfuscation method. It uses a custom construction using MD5 as a block cipher. Interestingly, we have not seen it observed elsewhere that this obfuscation method is unauthenticated, and thus implementations may be vulnerable to decryption oracle-type attacks.[2]

Implementation vulnerabilities have also been found in RADIUS implementations. For example, a TLS session resumption vulnerability discovered in 2017 could result in an authentication bypass in FreeRADIUS [27].

2.3 The MD5 Hash Function

The MD5 hash function was designed in 1991 by Rivest and described in RFC 1321 [40]. Although it was known to be weak by the mid-1990s, it continued to be widely used for decades, even after full collisions were demonstrated.

[2] In order to protect against password-based timing attacks, FreeRADIUS delays Access-Reject messages by one second by default. [16]

The structure of MD5. The MD5 hash function uses the Merkle-Damgård transform [15,31] to iteratively apply a compression function to blocks of the input message. Messages are processed in blocks of 512 bits, and the final hash output is the output of the final application of the compression function to the final block of message padding.

For the purposes of the attack we describe in this paper, the outcome of this design means that if we have two input messages M_1 and M_2 that result in an MD5 collision $\text{MD5}(M_1) = \text{MD5}(M_2)$, we can append any identical suffix S to these messages and the resulting strings $M_1 \| S$ and $M_2 \| S$ will still have colliding MD5 hashes $\text{MD5}(M_1 \| S) = \text{MD5}(M_2 \| S)$.

2.4 MD5 Collisions

The first pseudo-collisions for the MD5 compression function were found by den Boer and Bosselaers [20] in 1993. Dobbertin exhibited a "free-start" collision in 1996 [21]. Wang, Feng, Lai, and Yu published a full MD5 collision in 2004 [52]. This collision consisted of two fixed strings. Exhibiting such a collision suffices to demonstrate that the MD5 hash function is no longer cryptographically collision-resistant, but attacks on MD5 in real-world contexts generally require collisions on attacker-chosen prefixes.

The structure of this collision was adapted in a straightforward way to an arbitrary common prefix, resulting in a so-called *identical-prefix* collision: given a prefix P, the algorithm computes gibberish blocks G_1 and G_2 such that $\text{MD5}(P \| G_1) = \text{MD5}(P \| G_2)$. This can be used to, for example, produce different PDF documents with identical hashes or to produce so-called "hashquines" [2, 28, 49].

The MD5 collision attack that we exploit in this paper is called a *chosen-prefix* collision and was introduced in 2007 [47]. That is, given distinct prefixes P_1 and P_2, we wish to efficiently compute gibberish blocks G_1 and G_2 such that $\text{MD5}(P_1 \| G_1) = \text{MD5}(P_2 \| G_2)$. Having done so, as noted in Section 2.3, we can then append any fixed suffix S and the resulting messages still have colliding MD5 hashes: $\text{MD5}(P_1 \| G_1 \| S) = \text{MD5}(P_2 \| G_2 \| S)$.

Our attack is based on the improved chosen-prefix collision attack by Stevens et al. [48] that reduced the original cost from [47] for a chosen-prefix collision from 2800 core-days to 39 core-hours. Additionally, this improved attack allows the attacker to reduce the length of collision block gibberish but at a larger computational cost.

Stevens et al. [48] computed a chosen-prefix collision in 2009 that was only 204 bytes long in 28 hours on an array of 200 Sony PlayStation 3s, and used it to create a rogue TLS CA certificate. In the same paper the authors also computed a chosen-prefix collision suffix of only 11+64 bytes long at a cost of about $2^{53.2}$. The `hashclash` code implementing both attacks was made publicly available [46].

The authors appeared to have made their point about the importance of deprecating MD5, and it seems that little further public MD5 cryptanalysis has happened in the intervening 15 years. Nevertheless, despite these MD5 attacks, in 2016 TLS still supported MD5, which Bhargavan and Leurent [7] showed to be vulnerable to transcript collision attacks. They demonstrated an "online" attack in about 1 hour on a 48-core machine using a slightly modified version of `hashclash`.

2.5 Hashclash chosen-prefix attack

At a high level, the chosen-prefix collision attack as implemented in `hashclash` has two stages, called "birthday" and "near-collision". We follow the naming convention of `hashclash`, where the (4×32)-bit *intermediate hash value* (IHV) of some number of message blocks is the internal state of MD5 after processing those message blocks. The difference in the IHVs (called dIHV) between a pair of (partially-constructed) messages will be zero when we have a collision.

Birthday. In the first stage, the algorithm finds bits to append to both prefixes so that the first 32-bit word of dIHV is zero, and the last two 32-bit words of dIHV are equal.

These birthday bits are found using the van Oorschot and Wiener [50] parallel collision search with distinguished endpoints. A large number of *trails* are computed from random starting points. When there are colliding endpoints, birthday bits can be computed from the trail starting points. One stops when a found dIHV only needs B near-collision blocks, where the cost of the birthday stage depends on B, and ranges from 2^{37} for 9 blocks up to 2^{53} for 3 blocks. There are GPU and CPU implementations of this stage in `hashclash`.

Near-collision. The second stage iteratively reduces the number of bit differences in dIHV to zero by appending a sequence of *near-collision blocks*. For each near-collision block, one first constructs *differential paths*: a set of conditions on the bits of the message blocks and of the internal state that will result in the desired change to dIHV. These differential paths are constructed via a *forward* stage that works forward from the previous IHV values, a *backward* stage that works backward from the desired dIHV-change, and a *connect* stage that finds a path connecting a forward path to a backward path. Once a differential path and the corresponding conditions on message bits are known, a *find-collision* algorithm searches for a message block pair satisfying the bit conditions and checks whether they result in the desired changes to dIHV. This find-collision stage is embarrassingly parallel, but the forward, backward, and connect stages are not. These four near-collision stages are only implemented for CPUs.

After enough near-collision blocks, dIHV is reduced to zero, resulting an MD5 collision. Each near-collision cost is comparable to about 2^{35} MD5 compressions.

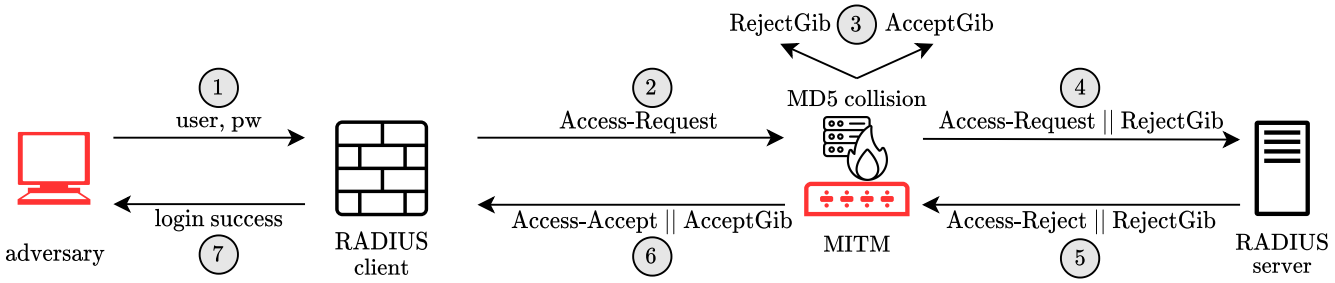

Figure 1: **Our attack flow.** Our adversary triggers an Access-Request with incorrect credentials from a legitimate RADIUS client (1), and then carries out a man-in-the-middle attack (2) and computes an MD5 hash collision (3) to inject a malicious Proxy-State attribute in the request (4). The hash collision allows the attacker to transfer the server-generated Response Authenticator from the legitimate Access-Reject response (5) to the attacker's desired Access-Accept response (6) to get authenticated or authorized by the RADIUS client (7).

3 Forging RADIUS Response Authenticators

In this section, we describe our protocol attack against RADIUS Response Authenticators.

3.1 Attack Model

The adversary in our attack wishes to trick a victim device into accepting a forged response from a RADIUS server.

A successful attack allows various forms of privilege escalation, depending on the access protected by RADIUS. For example, for a network router that uses RADIUS to authenticate administrative access to the device, such an attack would allow an attacker with network privileges to log in as an administrator without a password. For a victim NAS that uses RADIUS to authenticate network users to a protected VLAN or VPN, our attack would allow the adversary to access arbitrary privileged portions of the network without credentials.

We assume that the adversary does not know the shared secret between the RADIUS client and server.

Gaining Man-in-the-Middle Access. Our attack requires the adversary to have network access to act as a man-in-the-middle attacker on the connection between the victim device's RADIUS client and RADIUS server. Our attacker will need to be able to act as a full network man-in-the-middle who can read, intercept, block, and modify inbound and outbound network packets between the RADIUS client and server.

Such access to RADIUS traffic may happen through different mechanisms. Although sending RADIUS/UDP over the open internet is discouraged, this is still known to happen in practice [18, 38]. For internal network traffic, the attacker might initially compromise part of an enterprise network; such compromises appear frequently in news reports and security advisories [10, 12]. Even if RADIUS traffic is confined to a protected part of an internal network, configuration or routing mistakes might unintentionally expose this traffic. An attacker with partial network access may be able to exploit DHCP or other mechanisms to cause victim devices to send traffic outside of a dedicated VPN [33].

3.2 Attack Flow

We describe the steps of our attack as shown in Figure 1.

Step 1 (Login Attempt). We consider the simplest case of PAP authentication. The adversary causes the RADIUS client of a victim's network device to generate an Access-Request by, for example, entering the username and an arbitrary incorrect password at a login prompt for a privileged user.

Step 2 (Access-Request). The RADIUS client generates an Access-Request containing a one-byte ID, a 16-byte randomly generated Request Authenticator, and a collection of attributes including User-Name and User-Password containing the obfuscated incorrect password entered by the adversary. The client sends this Access-Request to the RADIUS server.

Step 3 (Collision). Since the adversary has network access, it is able to observe all of these values in the UDP packet. It will need the ID and Request Authenticator (`ReqAuth`).

If the adversary were to forward this Access-Request packet unmodified to the RADIUS server, it could then observe the format of the Access-Reject response generated by the server. This packet will contain a Response Authenticator that is calculated as $\text{MD5}(\texttt{Code} \,\|\, \texttt{ID} \,\|\, \texttt{Len} \,\|\, \texttt{ReqAuth} \,\|\, \texttt{Attributes} \,\|\, \texttt{Secret})$ as described in Section 2.1.

Once the adversary is ready to carry out the attack, it uses its man-in-the-middle network position to block or mangle the legitimate Access-Request packet so that the server does not receive it or discards it, and begins to generate a maliciously modified request to send to the server instead.[3]

[3] If the server receives both modified and unmodified requests, its behavior is implementation dependent. The modified packet may be treated as a

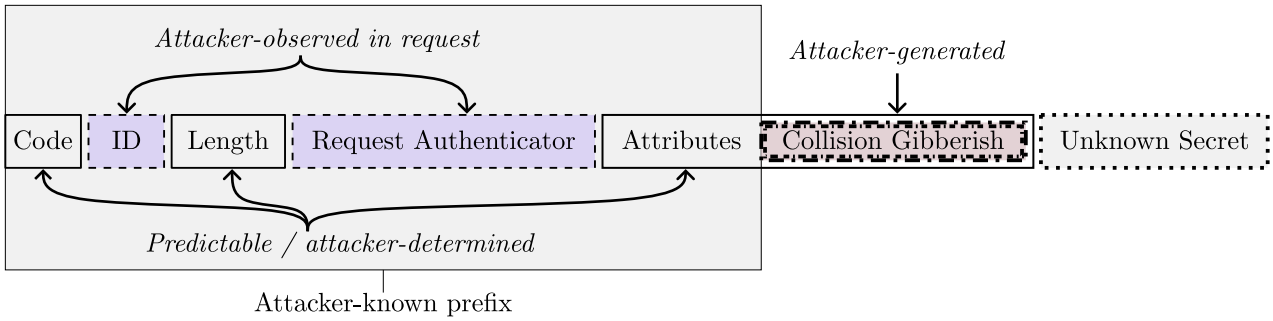

Figure 2: **Constructing Response Authenticator Preimages.** The attacker computes prefixes corresponding to the Response Authenticators from the Access-Reject packet it expects and Access-Accept it desires to forge, and carries out an MD5 collision to ensure the Response Authenticators will collide, without needing to know the shared secret. It hides the collision block gibberish in Proxy-State attributes that are guaranteed to be returned by the server.

Our adversary wishes to modify the Access-Request packet so that the Response Authenticator for the resulting Access-Reject packet is identical to the Response Authenticator for an Access-Accept packet. To do this, our adversary will compute a chosen-prefix MD5 collision. Our adversary does not know the value of the shared secret in the Response Authenticator computation, but this does not matter: it knows or can guess all of the other values that precede the Shared Secret, and as observed in Section 2.3, any identical suffix appended to an MD5 collision will still produce a collision. Our attacker will compute a collision for the Response Authenticator preimage up to the shared secret, and can then transfer the observed Response Authenticator sent from the legitimate server to the maliciously constructed colliding packet.

However, computing an MD5 collision generally involves appending hundreds of bits of unformatted collision block gibberish to the chosen prefixes. Fortunately for the attacker, the Proxy-State attribute of a RADIUS packet is an ideal place to hide this gibberish: any Proxy-State attributes included in an Access-Request must be returned unmodified by the server.

Our network attacker will calculate two prefixes for an MD5 collision with the structure shown in Figure 2. The first prefix is the start of the preimage of the Response Authenticator in the Access-Reject packet that it expects to receive, and ends with the header for a Proxy-State attribute (a code 33 and length `PSLen`): in the simplest case this would be

$\texttt{RejectPrefix} = \texttt{03} \,\|\, \texttt{ID} \,\|\, \texttt{Len} \,\|\, \texttt{ReqAuth} \,\|\, \texttt{33} \,\|\, \texttt{PSLen}.$

Any attributes expected in the reject packet can be included in this prefix. The second prefix is the start of the preimage of the Response Authenticator in the Access-Accept packet that it wishes to forge, and ends with a Proxy-State header:

$\texttt{AcceptPrefix} = \texttt{02} \,\|\, \texttt{ID} \,\|\, \texttt{Len} \,\|\, \texttt{ReqAuth} \,\|\, \texttt{33} \,\|\, \texttt{PSLen}.$

Computing a chosen-prefix collision on these two prefixes results in binary gibberish strings `RejectGib` and `AcceptGib` such that $\text{MD5}(\texttt{RejectPrefix} \,\|\, \texttt{RejectGib})$ equals $\text{MD5}(\texttt{AcceptPrefix} \,\|\, \texttt{AcceptGib})$. We have structured the prefixes so that `AcceptGib` and `RejectGib` will be interpreted as Proxy-State attributes.

Step 4 (Modify Access-Request). After computing the collision, the man-in-the-middle attacker modifies the Access-Request packet to include a Proxy-State attribute containing `RejectGib`. (If necessary, the attacker also strips the request of any Message-Authenticator attributes.) There is no other authentication, so the modification will be unnoticed. The adversary sends the modified Access-Request to the server.

Step 5 (Access-Reject). The server responds with an Access-Reject (or Access-Challenge or Access-Accept) that includes a Response Authenticator computed over the preimage shown in Figure 2 including the Shared Secret. The RADIUS protocol mandates the Proxy-State attributes containing `RejectGib` be included in the server's response, and thus in the Response Authenticator MD5 hash.

Step 6 (Forge Access-Accept). The adversary then verifies that the attributes present in the Access-Reject packet match its anticipated format for the collision. If so, it constructs the desired Access-Accept packet. Conveniently, the Access-Accept packet whose Response Authenticator corresponds to $\texttt{AcceptPrefix} \,\|\, \texttt{AcceptGib}$ is exactly the string $\texttt{AcceptPrefix} \,\|\, \texttt{AcceptGib}$ (see Step 3 above) with the Request Authenticator from the original request replaced with the Response Authenticator from the Access-Reject packet.

Step 7 (Login Success). The adversary then forwards the Access-Accept packet to the victim RADIUS client, who will validate the Response Authenticator against the received packet and shared secret and permit the adversary to log in.

retransmission, and receive a cached response. In FreeRADIUS, duplicate detection uses the packet length. If a server receives the original "good" packet, followed by the attacker's "bad" packet, the server may stop processing the "good" packet and process the "bad" packet instead. [16]

3.3 Adding Additional Attributes

An Access-Accept response may contain attributes specifying network configuration or access to particular resources like a VLAN. An attacker could forge arbitrary attributes in its Access-Accept by including them in the MD5 collision prefix; additional length in the corresponding Access-Reject (or Access-Accept) prefix can be covered by more Proxy-States.

3.4 Practical Challenges

There are several logistical challenges that we need to overcome in order to carry out this attack in practice. First, because the prefixes the attacker is using for the MD5 collision contain the single-byte request ID and random 16-byte Request Authenticator nonce, the attacker must compute the MD5 collision *online* after observing the victim's Access-Request and before the victim client's programmed session timeout. Realistic timeout lengths are generally in the range of 30 to 60 seconds. RFC 5080 suggests a default timeout length of 30 seconds [34]; 60 seconds is a common recommendation when using multi-factor authentication.

Second, the length of the collision must fit within the Proxy-State attribute. The MD5 hash function has a 64-byte block size; to fit within the 253-byte maximum length of a single Proxy-State attribute the collision can be at most three blocks long, plus additional padding and birthday bits. We develop a new technique that allows us to split a longer collision across multiple Proxy-State attributes by placing the two-byte header for additional Proxy-State attributes at desired locations during the collision computation.

3.5 Accounting Request Authenticators

In theory, a similar collision attack affects the construction of RADIUS Accounting Request Authenticators. This construction is also used for CoA-Request and Disconnect-Request [22]. However, the details of exploitation are different, and a practical attack may be more difficult. We did not attempt to implement this attack, but we detail the underlying cryptographic vulnerability.

As described in Section 2.1, RADIUS Accounting-Request packets include a Request Authenticator that is computed similarly to the Response Authenticator we exploit above, except with 16 null bytes in place of the random Request Authenticator from an Access-Request.

An attacker trying to log false accounting information might compute a chosen-prefix MD5 collision to find a colliding pair of Accounting-Request packets: one with the desired false accounting information, and one with true accounting information that the attacker can cause or expect the RADIUS Accounting client to send. As before, the attacker does not know the shared secret, but the packets will still collide when the shared secret is appended, so the correct request authenticator will be the same for both packets.

When the client tries to send the packet with true accounting information, the man-in-the-middle attacker copies the packet's Request Authenticator onto the forged accounting request and sends that instead. If the attacker has correctly predicted the true request, the collision will result in a valid Request Authenticator for the forged request.

Exploiting this would require an attacker to predict the exact content of the Accounting-Request packet. This may be complicated, as attributes include a variable-length Acct-Session-Id attribute that is chosen in an implementation-dependent way, as well as timestamps.

Furthermore, the attacker has much more limited control of the values appearing in the request. In this attack model, the attacker cannot, for example, cause the client to send arbitrary Proxy-State attributes: it needs to cause the client to include its desired collision block gibberish in a legitimate request by including it in an attribute that it can control.

One example of an attacker-controlled attribute might be User-Name. The attribute has a maximum length of 253 bytes; this is room for a three-block MD5 collision plus additional padding. As we note in Section 2.4, it is feasible to compute MD5 chosen-prefix collisions as short as birthday bits plus only a single block: 11+64 bytes in length. A User-Name attribute embedding a null character between a valid username and some collision gibberish could be parsed by a vulnerable implementation using standard C string functions as a normal username, as in the null prefix attacks against SSL/TLS certificates by Marlinspike and Kaminsky [29].[4]

An additional practical complication to this attack is that a chosen-prefix collision would only allow falsifying attributes that appear *before* the collision garbage: all bytes after the collision gibberish would need to be identical.

Since this theoretical attack against RADIUS accounting is likely more difficult to practically exploit than the attack against RADIUS authentication, we leave a practical evaluation to future work.

4 Optimizing MD5 Chosen Prefix Attacks

Our attack requires the adversary to forge the Access-Accept before the victim client times out. This means that our man-in-the-middle adversary must compute an MD5 collision within the timeout window, after seeing the Request Authenticator and ID sent by the victim RADIUS client.

As discussed in Section 6, clients have configurable timeout settings ranging from as little as five seconds to five minutes, with common defaults of 30 and 60 seconds.

When we began this research, running `hashclash` with the default parameters on a machine with an AMD 5950X

[4] A March 1, 1999 email to the IETF RADIUS mailing list documented such a vulnerability affecting Proxy-State: "I had to implement a configurable option to avoid NULs in my Proxy-State attributes in order to interoperate with older Livingston/Merit code (not sure about the latest stuff) since they NUL-terminated all string attributes internally." [3]

CPU and an NVIDIA RTX3080 GPU on two 22-byte prefixes produced a 512-byte collision with 25% probability within 85 minutes. This does not yield a practical attack: 490 bytes of collision gibberish does not fit the 253-byte maximum length of a Proxy-State attribute, and the running time was far longer than a reasonable client timeout range.

In order to mount our attack in practice, we needed to compute MD5 collisions much faster while also remaining compatible with the RADIUS message format. After our optimizations, we could compute a collision formatted as two Proxy-State attributes in under 5 minutes parallelized across our heterogeneous cluster. On the single machine above, our new attack finishes with 25% probability within 30 minutes.

This required large and small changes: we added new capabilities to `hashclash` to fix certain bytes in the collision, shifted some of the computation to precomputation, and optimized the code for speed, scaling, and parallelization latency. We discuss the most important changes below, and detail all of them in Appendix B. The code is available at [46].

4.1 Collisions with Infix Bytes

There is a tradeoff between the length of the collision gibberish and the cost of the birthday search phase: shorter collisions cost more. We wanted to be able to hide longer collision gibberish across multiple consecutive Proxy-State attributes. This requires periodically encoding the two-byte header starting a new attribute in certain near-collision blocks.

One solution achieving this is to choose a 32-bit word m_0 in `MD5Compress` containing the two-byte infix, which fixes the differential path for step $t = 0$, and let the forward stage start at $t = 1$. This unnecessarily reduces the freedom in differential paths and may increase the near-collision search cost.

Instead, our optimal solution is to find all valid differential paths over step $t = 0$ with minimum number of bit conditions that guarantee the two-byte infix. We iterate over all possible m_0 containing the two-byte infix to obtain many valid differential paths for step $t = 0$. Then we simplify each path by attempting to remove each bit condition on the output Q_1 of step $t = 0$ and checking if the result still guarantees the two-byte infix sequence. If so, the bit condition can be safely removed. This approach may generate minimal differential paths multiple times, so we remove any double occurrences.

4.2 Precomputation

Since we are optimizing for the online running time once the prefixes are known, we benefit from shifting some of the computation into a precomputation phase.

For the "backward" phase, we precompute the set of upper paths for all possible message differences $m_{11} = \pm 2^i$ for $i = 0, \ldots, 31$. As the last four steps depend on the actual dIHV in the online attack, the upper paths are only computed over steps $t = 16, \ldots, 34$ that are the same for all possible dIHV.

Due to the partial precomputation of upper paths, Connect will only output differential paths over steps $t = 0, \ldots, 34$. We extended the differential path helper tool to glue any found connected paths to the target upper path over steps $t = 35, \ldots, 63$ to obtain the required full differential paths.

Moreover, we extended the differential path helper tool to negate a set of differential paths. This allows us to convert the precomputed set for $m_{11} = +2^i$ to the desired precomputed set for $m_{11} = -2^i$ for free.

4.3 Cluster Parallelization

We have access to a cluster of 47 CPU machines, with one machine with four GPUs. (We give specifications in Section 4.4.) We developed a number of techniques to parallelize `hashclash` across our existing computing resources. Our machines are clustered together with a shared filesystem using GlusterFS and ZFS, and use Slurm for job management.

File I/O. For communication between processes, `hashclash` uses the file system, which creates a bottleneck when distributing on a cluster. We reduced latency in the code and used in-memory tmpfs filesystems shared via NFS, creating a primitive form of remote direct memory access.

Thread Orchestration and Parallelization. We modified several stages of the algorithm to partition input and output spaces. For example, we extended the "forward" phase to partition the output into N sets and save them concurrently instead of storing all output in one file single-threaded. This might sacrifice some quality for speed: some very good paths might not be found using this approach.

Birthday. The birthday phase has a mode that partitions the trail space over multiple machines. This works best with powerful GPUs. Since we only have one machine with four Tesla T4 GPUs, we instead used our CPU machines as controllers. We introduced a generator mode that uses the four GPUs to generate trails that are processed by the controllers. We use Slurm to start and orchestrate the CPU and GPU nodes.

4.4 Experimental results

We ran our experiments on a heterogeneous cluster with 31 Intel Xeon CPU E5-2699 v4, 14 Intel Xeon CPU E5-2680 v3, one Intel Xeon CPU E7-4860 v2, and one GPU machine with an Intel Xeon CPU E5-2673 processor and four Tesla T4 GPUs. Most machines have 512 GB of memory but none of our operations used significant amounts of RAM. We note that most of these machines date from 2014–2017.

Figure 3 shows the runtime distribution of our MD5 chosen-prefix collision attack. We ran 110 full attacks and aborted 42.7% before they found a collision because they were not

on a good trajectory. Since timings for birthday search and near-collisions are independent, we take their measured times separately and compute the direct sum of all possible combinations. According to Figure 3 we expect 2% of the successful runs to finish before 240s and 16% before 300s.

Our adversary can cope with the failure rate and randomized running time of the attack by triggering multiple requests until one succeeds.

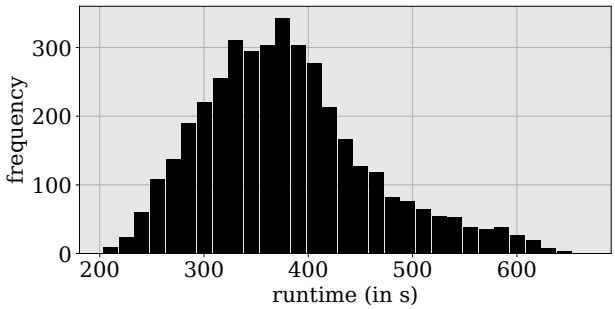

Figure 3: Histogram of the runtime of our attack as direct sum between birthday runtimes and near-collisions runtimes.

4.5 Prospects for further improvement

While we have been able to reduce the online running time for our MD5 chosen-prefix attack from hours down to minutes, this should be interpreted as a generous upper bound for the true cost of such collisions, because of the limits on our computational resources. Newer CPUs than the seven to ten year old machines we have access to would likely provide minutes of improvement, as would optimizing cache locality.

Access to more and faster GPUs would reduce the time for the birthday stage and/or reduce the number of near-collision blocks, reducing time for the near-collision stage.

Reimplementing `hashclash` in hardware, for example on FPGAs (Field Programmable Gate Arrays) or ASICs (Application-Specific Integrated Circuits) would likely improve the running time by a factor of ten to a hundred.

It would be eminently feasible to run this attack on cloud resources. Amazon EC2 lists the on-demand price of a `c7a.48xlarge` instance with 192 vCPUs at $9.85/hour, and the price of a `g6.48xlarge` instance with 192 vCPUs and 8 NVIDIA L4 GPUs at $13.35/hour. It would cost around $50/hour to exceed our computing capacity, and in principle one could scale to many more machines.

We did not pursue this avenue further for two reasons. First, based on previous experience the cost of simply implementing and debugging an attack in the cloud that requires launching hundreds of dollars an hour of computing instances can quickly reach tens of thousands of dollars. Second, achieving further gains would require us to more substantially rearchitect `hashclash`. We hope that the reader is already convinced that MD5 is exploitable.

4.6 Comparison to offline brute force

For the six-block collision we chose, the computational cost of our online collision attack is dominated by the birthday phase, with a complexity of around $2^{43.2}$, and is mostly done using GPUs. Comparing this cost to the cost of the offline brute force attack to recover shared secrets discussed in Section 2.2, this is somewhat less than the cost to brute force a 7-character shared secret. Current recommendations prescribe a minimum shared secret length of at least 24 octets in order to protect against offline brute force attacks [18]. Unfortunately, while it is straightforward to mitigate brute force attacks against the shared secret by generating longer shared secrets uniformly at random, this has no effect on the cost of our collision attack.

5 Attack Evaluation

We carried out our attack against several settings of client and server software.

5.1 MITM Implementation

We implemented a man-in-the-middle script in Python using the pyrad library to parse and format RADIUS packets, and our modified version of the `hashclash` software.

Our script implements the attack described in Section 3. It receives an Access-Request from the client, guesses the format of the Access-Reject packet that will be received from the server, then runs our modified version of `hashclash` on the corresponding Response Authenticator prefixes for Access-Reject and Access-Accept packets to compute a six-block MD5 collision formatted into two Proxy-State attributes. Our script appends the two Access-Reject Proxy-State attributes to the Access-Request, strips any Message-Authenticators, and forwards it to the server. After intercepting the Access-Reject response, we construct the Access-Accept packet, copy the Response Authenticator from the Access-Reject received from the server, and forward it to the client.

5.2 FreeRADIUS client and server

The FreeRADIUS website states that “FreeRADIUS is the most widely used RADIUS server in the world. It powers most major Internet Service Providers and Telecommunications companies world-wide and is one of the key technologies behind eduroam, the international Wi-Fi education roaming service. It is the RADIUS server used by all Cloud Identity providers and is embedded in products from network equipment vendors and token card manufacturers.” [35]

We installed FreeRADIUS 3.2.3 on a Linux machine running Ubuntu 23.10. We left the default configuration as is, which uses UDP as standard transport protocol. There is an option to require a Message-Authenticator attribute in an access request; this defaults to “no” with a comment that “Old-style

clients do not send a Message-Authenticator in an Access-Request. RFC 5080 suggests that all clients SHOULD include it in an Access-Request."

We then sent a PAP authentication request with the FreeRADIUS client `radclient` for a valid user with an incorrect password via our MITM to the RADIUS server. We set the client timeout to 10 minutes.

Neither `radclient` nor FreeRADIUS include a Message-Authenticator attribute by default. FreeRADIUS will include a Message-Authenticator in proxied Access-Request packets.

FreeRADIUS returned the Proxy-State attributes we included in the request packet. Finally, `radclient` accepted our forged Access-Request without complaining about the Proxy-State attributes.[5]

The `radtest` client does include a Message-Authenticator attribute in its requests, but stripping it has no consequences for the MITM attack with the default settings of FreeRADIUS.

5.3 Okta RADIUS Integration

The Okta authentication service supports RADIUS authentication via a custom agent, allowing customers to offload authentication and MFA support to their service.

We configured a test environment on Ubuntu 22.04 emulating the recommended default setup for such a deployment. In our setup, we ran a TinyRadius server that communicates with the Okta RADIUS Server agent. Within a customer organization's network, the Okta RADIUS Server agent communicates over UDP port 1812. The Okta RADIUS agent then makes requests over TLS to Okta's remote servers, and proxies the responses back to the TinyRadius server, which then communicates the responses to the RADIUS client. In this setup, the TinyRadius server is taking the place of a local authentication server that communicates with Okta.

Okta supports several multi-factor authentication mechanisms. Customers can choose different RADIUS authentication mechanisms depending on their chosen MFA methods: PAP, EAP-TTLS, and EAP-GTC. We used PAP authentication, configured so that the User-Password plaintext contained both the user's static password and MFA passcode (comma-delimited); MFA can also be configured as an Access-Challenge.

We used `radclient` to make a PAP authentication request over UDP to Okta via our man-in-the-middle script using the same `radclient` call as above. Since the User-Password field includes the MFA token value, we entered an arbitrary incorrect password.

The only difference to the setup described above in Section 5.2 is that the Okta servers returned a Reply-Message attribute with a welcome or error message, which we needed to include in our MD5 collision prefixes. No Message-Authenticator attributes were returned with the responses. Our injected Proxy-State attributes were successfully returned by the server as the last attributes in our request. Our MITM attack successfully forged an Response Authenticator for an Access-Accept in under 10 minutes that `radclient` accepted.

5.4 Cisco ASA Firewall

We configured a Cisco ASA 5505 firewall device to use RADIUS as an Authentication-Authorization-Accounting (AAA) server and pointed it to our FreeRADIUS server for authentication. The firewall supports using RADIUS to authenticate users for logging into the device by serial console, giving access to VPN tunnels, and using various services such as Telnet, FTP, and HTTPS. The Cisco ASA has a configurable timeout with a maximum of 300 seconds. We successfully carried out our attack against the Cisco device and forged an Access-Accept message that it accepted within the 300 second timeout.[6] The ASA RADIUS client did not object to our Proxy-State attributes.

5.5 PAM Radius

The pluggable authentication module (PAM) named `pam_radius`[7] "allows any Linux, OSX or Solaris machine to become a RADIUS client for authentication". For example, it can be used to allow programs like `sshd`, `su`, `sudo`, and `login` to query a RADIUS server instead of the local `/etc/passwd` and `/etc/shadow` when authenticating users. It speaks RADIUS over UDP and does not use EAP.

One barrier to demonstrating a practical attack against `pam_radius` is that the maximum configurable timeout is 60 seconds. While it may be possible to exploit retries to lengthen the attack window, for the purpose of expediency we modified the source code to change the maximum timeout from 60 seconds to 600 seconds and rebuilt from source.

We installed our custom build of `pam_radius` on a Linux machine running Ubuntu 22.04, and configured the PAM subsystem to make `su` use RADIUS for authentication.

We pointed the module at the FreeRADIUS server we configured above, via our MITM script. The `pam_radius` module accepted our forged Access-Accept packet (without warnings about the extraneous Proxy-State attributes), allowing us to successfully run `su` with an incorrect password.

5.6 EAP-TLS

The TLS in EAP-TLS protects the EAP traffic but not the RADIUS packets that carry the EAP traffic, which are still

[5] Alan DeKok notes that `radclient` is not intended to complain about anything other than shared secret mismatch, but that he is not aware of any RADIUS client that will take any action when it receives an unexpected Proxy-State attribute. [16]

[6] The most challenging part of the attack was configuring the ASA and configuring our own network to forward UDP packets over TCP to our cluster.

[7] https://github.com/FreeRADIUS/pam_radius

transmitted over UDP in the clear. In theory, depending on implementation decisions made by the RADIUS client and server, our attack could work against RADIUS even when clients use EAP-TLS for authentication. We found that `hostapd` as an EAP / RADIUS client was not vulnerable, but this does not rule out other clients. Similar issues may affect other EAP methods.

Specifically, per RFC 3579 [8], all RADIUS packets with an EAP-Message need a valid Message-Authenticator, and ordinarily an Access-Accept after successful EAP authentication would contain an EAP-Message indicating EAP success. The Message-Authenticator is an HMAC-MD5 of the packet, which we are not able to forge. But the RFC is less clear on how a client should handle an Access-Accept packet at the end of an EAP conversation that has neither an EAP-Message nor a Message-Authenticator; this situation should not happen with a well-behaved server. The RFC states: "If no EAP Message attribute is included within an Access-Accept or Access-Reject, then the peer may not be informed as to the outcome of the authentication, while the NAS will take action to allow or deny access" [8].

In particular, the RFC does not specify whether a client using EAP for authentication should check for a valid Message-Authenticator in an Access-Accept without an EAP-Message. Even when EAP (including EAP-TLS) is expected to be used, a client that does not perform this check would be vulnerable to our attack. Our colliding packets would contain neither EAP-Message nor Message-Authenticator attributes; the Access-Accept should make the client allow access, even without the EAP-Message. This assumes the attacker can cause the server to produce an Access-Reject without a Message-Authenticator, perhaps by trying non-EAP authentication.

While we found that `hostapd` does check for a Message-Authenticator even when no EAP-Message is present, other RFC-conforming clients might not.

A full exploit of such an attack would depend on the broader protocol context. If, as in WiFi discussed in Section 2.1, further communication requires session keys that would have been negotiated via EAP, a forged Access-Success would not be enough for a full attack.

6 Impact

Although alternative transports were proposed a decade ago, RADIUS/UDP remains common in real-world deployments.

Unfortunately, we are not able to use network scanning to provide a representative picture of RADIUS deployments. Many RADIUS servers will be on internal networks. Even for external-facing servers, RADIUS hosts are identified by IP address and servers only accept packets from allowed addresses. The RADIUS RFC specifies that servers should drop requests for hosts that they have not been pre-configured to have a shared secret with. Our scanning host would not be whitelisted by properly configured servers, so an internet-wide scan would thus only turn up misconfigured servers. Additionally, since it is a UDP-based protocol with no handshake before a login request, we cannot do TCP SYN scanning and would need to scan using a well-formed UDP Access-Request, which would appear as an attack to network administrators.

This does not affect the adversary in our attack model, who intercepts traffic between a legitimate client and server.

We examine case studies of real-world deployments instead. There is little documentation about RADIUS deployments in the literature. These case studies are intended to provide concrete instantiations for the factors impacting a practical attack, as well as to illustrate the complex impacts.

6.1 Case Study: A RADIUS Deployment

We give details about RADIUS usage in a large organization whose network administrators were willing to share details of their configuration with us. This organization has around 80,000 affiliated individuals, and uses RADIUS for authentication to the VPN and Wi-Fi network. Their system sees around 90,000 active RADIUS connections at a time, and 300,000 clients over a typical five-day period.

When users connect to the VPN, the end user VPN client connects to the VPN controller, which offloads AAA (Authentication, Authorization, and Accounting) to a Cisco Identity Services Engine (ISE). The Cisco ISE makes RADIUS/UDP requests to a set of load balanced RADIUS servers using PAP mode for authentication. When using multi-factor authentication, the RADIUS requests are proxied to the Duo service. Within the organization, the RADIUS/UDP traffic passes over a VLAN with restricted access. The timeout for the ISE proxy is set to 60 seconds.

Wi-Fi authentication is also done over RADIUS/UDP, using EAP-TLS authentication. The timeout is set to 5s, with 3 retries. Password authentication is done via Active Directory, with passwords sent via NTLMv2.

This organization uses TACACS+ to authenticate network administrators to the routing infrastructure.

Evaluation. In this setting, the VPN authentication setup would be vulnerable to our attack. However, the fact that RADIUS traffic passes over a restricted-access VLAN means that exploitation would likely be an involved process.

6.2 Case Study: Network Infrastructure

RADIUS is essentially universally supported as an access control method for routers, switches and other network infrastructure; RADIUS and TACACS+ are the most commonly used protocols for admin access.

Public security advisories and leaked documents make it clear that compromising routers and switches is a priority for state-sponsored attackers. A 2022 US CISA announcement details a collection of state-sponsored attacks targeting

Cisco and Juniper equipment used by telecommunications and network service providers via credentials obtained from a RADIUS server [11]. Documents leaked by Edward Snowden also highlight the compromise of routers and switches as a priority for the NSA [56].

Evaluation. In this setting, an attacker who is able to access portions of the network that carry RADIUS traffic could exploit our attack to gain privileged access to all of the routers in an organization.

6.3 Case Study: eduroam

The eduroam network is a coalition of educational institutions who share Wi-Fi access via a linked hierarchy of RADIUS servers. In order to participate in eduroam, an institution sets up a RADIUS server. Internet access requests from users who are roaming at participating institutions are proxied via this network of RADIUS servers back to the home institution.

The Internet Draft for RADIUS over (D)TLS [38] documents the RADIUS deployment in eduroam as an example of RADIUS/TLS in "more than a dozen" national branches. The document states as of October 2023, "RADIUS/UDP continued to be used by a majority of country deployments despite its significant security issues." This implies that although RADIUS/TLS "does work, and scales," most eduroam RADIUS traffic is transmitted over the open internet via UDP.

Because eduroam uses 802.1X, it uses EAP within RADIUS for authentication, most commonly EAP-TLS, EAP-TTLS, or PEAP. We spoke to network administrators at an institution participating in eduroam; they told us that the eduroam RADIUS clients are configured with a 15s timeout with three retries.

Evaluation. An attacker may be able to forge an Access-Accept without an EAP-Message as described in Section 5.6. However, as described in Section 2, in the context of 802.1X, further communication relies on a master key that would normally be negotiated during the EAP session, and an attacker who bypasses EAP authentication would not have such a key.

7 Mitigations

Our recommended mitigations were developed in conversation with vendors. Our suggested temporary mitigations comply with the standard and preserve backward compatibility with unpatchable legacy hardware. For hypothetical mitigations that break backward compatibility, it would be better to redesign the entire protocol.

7.1 Short-Term Fix: Message-Authenticator

The optional Message-Authenticator attribute specified in RFC 2869 [54] computes an HMAC-MD5 over the entire packet. Packets that fail this integrity check are specified to be silently discarded. Although the MD5 hash function is no longer collision resistant, this is not known to imply an attack against HMAC-MD5, and HMAC-MD5 has been proven to be secure under the assumption that the compression function is a PRF, removing any reliance on collision resistance [5].

Our attack would be impossible if valid Message-Authenticator attributes were required on all packets, since the HMAC-MD5 would be unforgeable without the shared secret. As specified in RFC 2869 [54], Message-Authenticators are only required for packets containing EAP-Message attributes; RFC 5080 [34] states that clients SHOULD include Message-Authenticators on requests, but does not mandate it.

We learned from vendors that nearly all RADIUS servers are based on one of five implementations: FreeRADIUS, Radiator, Cisco, Microsoft, and Nokia. Before our disclosure, none required Message-Authenticators on non-EAP requests by default. The only RADIUS server we are aware of that did require them by default was OpenBSD's `radiusd`. Among clients, `hostapd` was the only implementation that required Message-Authenticators on non-EAP responses. The client implementations of FreeRADIUS and NPS could be configured to require them but did not do so by default; we are not aware of any other client with such an option.

Requiring Message-Authenticators. Our recommended short-term mitigation for the attack is for clients and servers to always send and require Message-Authenticator attributes for *all* requests and responses. For Access-Accept or Access-Reject responses, the Message-Authenticator should be included as the first attribute. See [19] for specific implementation details. This mitigation has been implemented by all RADIUS implementations that we are aware of.

This measure breaks compatibility with old implementations that may not include Message-Authenticators in requests or responses. However, unlike other options, it is not a fundamental change to the protocol and can be adopted as a fairly simple patch to clients and servers.

Attacks on Partial Enforcement. Unfortunately, it is not enough for senders to always include a Message-Authenticator if the receiving party does not require its presence. We give two example attacks allowing an attacker to circumvent these incomplete mitigations.

For Access-Request packets, the attacker can simply strip a Message-Authenticator sent by a client if it is not required by the server. This is because there is no other authentication of the packet contents. Once the attacker has removed the Message-Authenticator, the request can be modified as desired without being detected.

In the other direction for Access-Accept and Access-Reject responses, a man-in-the-middle attacker cannot simply strip this attribute from the packet as for requests, because the Message-Authenticator attribute is included in

the Response Authenticator. However, we observed that the Message-Authenticator attribute was typically the last attribute in the packet in implementations we examined. If the Message-Authenticator is not the first attribute in the packet then our man-in-the-middle attacker can hide it in a Proxy-State or other attribute by crafting a malicious prefix to end with a Proxy-State header, and simply copy the bytes of the Message-Authenticator into the Proxy-State after the collision. The receiving client will interpret this packet as a valid packet without a Message-Authenticator.

To mitigate this attack, we recommend servers always include a Message-Authenticator as the *first* attribute in any response. This appears to prevent the above attack even if a client does not validate the HMAC-MD5. This is because the value of the Message-Authenticator will be unpredictable, and thus the attacker will not be able to guess its value to include in the collision prefix or abuse other attribute headers to swallow the unpredictable value in the collision.

Practical considerations and interoperability issues for deploying these mitigations are discussed further in [19]. Future versions of [18] are expected to mandate this behavior for clients and servers.

7.2 Long-term fix: RADIUS over (D)TLS

The long-term solution to the vulnerabilities we describe in this paper is to use RADIUS inside of an encrypted and authenticated channel that offers modern cryptographic security guarantees. We expect future work in the IETF to formally deprecate RADIUS/UDP and standardize RADIUS over (D)TLS.

Transport security for RADIUS has been defined in the experimental RFC 6614 [53], specifying RADIUS over TLS, and RFC 7360 [17], specifying RADIUS over DTLS.

The IETF RADIUS EXTensions working group has a current Internet Draft to update and modernize the security recommendations for (D)TLS Encryption for RADIUS [38]. It is not yet on the standards track.

Unfortunately, although the proposals for RADIUS over TLS have existed for a decade, many systems continue to use RADIUS/UDP. Deploying TLS and maintaining a public-key infrastructure continues to be a much more involved process for system administrators than simply using UDP; many of the challenges are documented in the (D)TLS Internet Draft [38].

7.3 Partial Mitigation: Management VLAN

A current best practice for RADIUS/UDP traffic is to expose it only to a restricted-access management VLAN within an organization. While this reduces the attack surface and is certainly preferable to exposing UDP traffic to a broader network or the open internet, there may still be vulnerabilities in case of a network misconfiguration or attacker compromise of this portion of the network.

This approach is also at odds with the US Executive Branch Office of Management and Budget's 2022 memo, which envisions moving to systems that do not rely on network separation for security: "A key tenet of a zero trust architecture is that no network is implicitly considered trusted" [55].

7.4 Non-Mitigation: Decreasing timeouts

It is tempting to hope that simply setting a client timeout below our reported MD5 collision running times would defend against our attack. We believe this should not be done: it decreases usability and does not protect against our attack.

Our MD5 collisions were computed after applying some optimizations to a 15-year-old proof-of-concept codebase, which we are running on CPUs mostly dating from seven to ten years ago, because these are the resources we have access to. An adversary with a budget for professional engineering would be able to decrease the computational cost of the collision by a factor of tens to hundreds (cf. Section 4.5).

The most common timeout in practice is 30 seconds [16], and 60 seconds is commonly recommended when multifactor authentication is involved (for example, [23]) as shorter timeouts could be problematic for real users.

7.5 Non-Mitigation: TACACS+ or Diameter

RADIUS is not the only protocol to suffer from the types of security issues that we outline. TACACS+ is a popular (TCP-based) administrator login protocol for switches that also does not meet modern cryptographic security standards. RFC 8907 [14] was published in September 2020, and explicitly mandates that TACACS+ be used with a secure transport:

> With respect to the observations about the security issues described above, a network administrator MUST NOT rely on the obfuscation of the TACACS+ protocol. TACACS+ MUST be used within a secure deployment; TACACS+ MUST be deployed over networks that ensure privacy and integrity of the communication and MUST be deployed over a network that is separated from other traffic.

There is a current draft for TACACS+ over TLS 1.3 [13]. Much like RADIUS, however, TACACS+ is still most commonly used over insecure transports [16].

A suitable countermeasure would be to mandate transport security for RADIUS in an analogous fashion.

Diameter [24] was initially designed as a successor to RADIUS, although it never replaced RADIUS for many common use cases. It is used in 3G+ networks, and is generally only supported in large NAS equipment used by bigger ISPs and telecommunications providers; consumer or enterprise-grade equipment typically only supports RADIUS [16].

Although Diameter was intended to replace RADIUS, the protocol itself offers no security when used over TCP. As a result, RFC 6733 suggests that Diameter messages should be secured using TLS or DTLS; 5G has replaced Diameter with signaling over HTTP/2 [30]. The US government has described exploits against Diameter targeting mobile users [10].

7.6 Non-Mitigation: Random shared secrets

Organizations can protect against dictionary attacks on the shared secret by picking random shared secrets of sufficient length, as the runtime of such an attack grows exponentially with the entropy of the secret. For example, [18] recommends shared secrets with at least 96 bits of entropy, so an offline dictionary attack would involve on the order of 2^{96} MD5 compressions. However, as our attack does not try to brute-force the shared secret, choosing a strong shared secret does not affect the runtime of our attack.

7.7 Non-Mitigation: MFA

Multi-factor authentication (MFA) is not a mitigation either. Our attack largely bypasses the user authentication mechanism, and forges the accept response from the server's reject. MFA may be supported through multiple mechanisms within the RADIUS protocol, including authentication protocols like PAP that are vulnerable by default to our attack.

For example, when using Okta, a user's password and passcode can be entered together (as a single comma-delimited string) as the User-Password attribute in an Access-Request using the RADIUS PAP protocol. We carried out a successful proof of concept attack against Okta with MFA enabled in this setting, detailed in Section 5.3.

7.8 Non-Mitigation: Rejecting Proxy-States

Our forged Access-Accept packets contain Proxy-State attributes that the client is not expecting. However, having the client discard packets with unexpected Proxy-States does not mitigate the vulnerability. First, such a mitigation would only apply to a NAS; the Proxy-State attribute is actually used by RADIUS server proxies and thus difficult to remove without breaking functionality [16].

Even if NAS clients rejected unexpected Proxy-State attributes, it would be possible to craft colliding packets where the Access-Accept has the collision gibberish in a different attribute such as Vendor-Specific or Reply-Message that is likely to be accepted; the client does not need to support or attempt to interpret the garbage attribute to be vulnerable.

The colliding Access-Reject packet would still use Proxy-State attributes, as the server is guaranteed to include Proxy-State attributes unchanged in an Access-Reject. For simplicity our implementation uses Proxy-States in both colliding packets, as no RADIUS client we tested complained about the unexpected Proxy-State.

7.9 Non-Mitigation: Replacing MD5

It is tempting to think that simply replacing MD5 in the Response Authenticator with a secure hash function like SHA-2 or SHA-3 might be a short-term mitigation against our attacks. However, since the RADIUS protocol does not provide for any cryptographic agility, such a change would be incompatible with all existing implementations, and thus be equivalent to requiring a new protocol. Given the other security and privacy concerns with the rest of RADIUS, it would be better at that point to redesign the entire protocol or transport.

8 Conclusion

Beyond the RADIUS protocol itself, our work raises some broader issues in cryptographic security.

Over the past couple of decades, there has been a concerted effort in both academia and industry to deploy secure encrypted protocols on the internet.

However, the world of standardization bodies, and the relationship between standards and real-world deployment can be extremely complex and require large investments of time and money. This process relies on the economic interest of companies to fund employees to participate; long-term maintenance of protocols, working groups, and open-source software is subject to the shifting tides of employment and interests. This contributes to the current situation where vital portions of our network infrastructure are essentially maintained as a labor of love by a small number of dedicated individuals. Academic researchers may in some cases even have negative incentives from their university employers to do practical work that does not immediately result in academic publications.

That said, joint efforts in recent years between cryptography researchers and industry to develop and standardize provably secure versions of protocols like TLS and secure messaging have been successful. Given the enormous amount of effort put into securing these protocols it is surprising that a protocol as ubiquitous as RADIUS has received so little cryptanalytic attention over the years. TLS may be the charismatic megafauna of cryptographic protocol research, but in order to actually secure our infrastructure we need to analyze and secure the entire universe of enterprise security that academic cryptographers have little to no visibility into or insight in.

Acknowledgments

We are grateful to Alan DeKok and Heikki Vatiainen for extensive discussions and helpful feedback. We would like to thank Jonathan M. Smith, Richard Barnes, Roman Danyliw, Vijay Sarvepalli, Stefan Savage, Geoff Voelker, Jim Madden,

Crys Harris, and Aaron Schulman for enlightening conversations and logistical assistance. N.H., M.H., and A.S.'s funding includes gifts from Google and Qualcomm.

References

[1] Siemens AG. Setting up role-based access control for siemens digital grid products. https://cache.industry.siemens.com/dl/files/182/109759182/att_956446/v1/APN-051_Setting_up_RBAC_for_Siemens_Digital_Grid_Products.pdf, 2016.

[2] Ange Albertini and Marc Stevens. Hash collisions and exploitations, 2019. https://github.com/corkami/collisions.

[3] CERIAS Information Security Archive. http://ftp.cerias.purdue.edu/pub/doc/network/radius/archive/.

[4] GSM Association. GPRS roaming guidelines (version 10). https://www.gsma.com/newsroom/wp-content/uploads/IR.33-v10.0.pdf, July 2017.

[5] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision resistance. *Journal of Cryptology*, 28(4):844–878, October 2015.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[7] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In *NDSS*. The Internet Society, 2016.

[8] Pat R. Calhoun and Dr. Bernard D. Aboba. RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP). RFC 3579, September 2003.

[9] CISCO. Configuring security on the GGSN. https://www.cisco.com/c/en/us/td/docs/ios/12_4t/mw_ggsn/configuration/guide/ggsnsec.pdf.

[10] Joseph Cox. Cyber official speaks out, reveals mobile network attacks in U.S. https://www.404media.co/cyber-official-speaks-out-reveals-mobile-network-attacks-in-u-s/, May 2024.

[11] Cybersecurity & Infrastructure Security Agency. People's Republic of China state-sponsored cyber actors exploit network providers and devices. https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-158a, June 2022.

[12] National Cybersecurity and Communications Integration Center. The increasing threat to network infrastructure devices and recommended mitigations. https://www.cisa.gov/sites/default/files/2023-01/ar-16-20173.pdf, August 2016.

[13] Thorsten Dahm, John Heasley, dcmgash@cisco.com, and Andrej Ota. TACACS+ TLS 1.3. Internet-Draft draft-ietf-opsawg-tacacs-tls13-08, Internet Engineering Task Force, May 2024. Work in Progress.

[14] Thorsten Dahm, Andrej Ota, D.C. Medway Gash, David Carrel, and Lol Grant. The Terminal Access Controller Access-Control System Plus (TACACS+) Protocol. RFC 8907, September 2020.

[15] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 416–427. Springer, Heidelberg, Germany, August 1990.

[16] Alan DeKok. Personal communication.

[17] Alan DeKok. Datagram Transport Layer Security (DTLS) as a Transport Layer for RADIUS. RFC 7360, September 2014.

[18] Alan DeKok. Deprecating Insecure Practices in RADIUS. Internet-Draft draft-ietf-radext-deprecating-radius-00, Internet Engineering Task Force, November 2023. Work in Progress.

[19] Alan DeKok. RADIUS and MD5 collision attacks, 2024. https://networkradius.com/assets/pdf/radius_and_md5_collisions.pdf.

[20] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of MD5. In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 293–304. Springer, Heidelberg, Germany, May 1994.

[21] Hans Dobbertin. Cryptanalysis of MD5 compress. http://www.cs.ucsd.edu/users/bsy/dobbertin.ps, May 1996.

[22] Gopal Dommety, Murtaza Chiba, David Mitton, Mark Eklund, and Dr. Bernard D. Aboba. Dynamic Authorization Extensions to Remote Authentication Dial In User Service (RADIUS). RFC 5176, January 2008.

[23] Duo. Duo two-factor authentication with RADIUS and primary authentication. https://duo.com/docs/radius.

[24] Victor Fajardo, Jari Arkko, John A. Loughney, and Glen Zorn. Diameter Base Protocol. RFC 6733, October 2012.

[25] GE. GE multilin UR family brochure. https://www.gevernova.com/grid-solutions/products/brochures/Multilin-UR-Family-brochure-EN-12657I-LTR-2020-R002.pdf, 2020.

[26] Joshua Hill. An analysis of the RADIUS authentication protocol. https://www.untruth.org/~josh/security/radius/radius-auth.html, November 2001.

[27] Pavel Kankovsky. CVE-2017-9148 FreeRADIUS TLS resumption authentication bypass. https://seclists.org/oss-sec/2017/q2/422, June 2017.

[28] Mako. A PDF that shows its own MD5. In Manul Laphroaig, Melilot, Evan Sultanik, Jacob Torrey, Ange Albertini, Philippe Teuwen, and sundry others, editors, *PoC||GTFO*, chapter 12, pages 56–59. Tract Association of PoC||GTFO and Friends, 2017.

[29] Moxie Marlinspike. Null prefix attacks against SSL/TLS certificates. https://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-PAPER1.pdf, July 2009.

[30] John Mattsson. Email to radext working group. https://mailarchive.ietf.org/arch/msg/radext/Zcuud3GyG221DXnPcvJ2TugwGu0/, May 2024.

[31] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[32] Karl Meyer. eduroam hits a new record—7.5 billion authentications in 2023. https://connect.geant.org/2024/01/23/eduroam-hits-a-new-record-7-5-billion-authentications-in-2023, January 2024.

[33] Lizzie Moratti and Dani Cronce. Tunnelvision (CVE-2024-3661): How attackers can decloak routing-based VPNs for a total VPN leak. https://www.leviathansecurity.com/blog/tunnelvision.

[34] David B. Nelson and Alan DeKok. Common Remote Authentication Dial In User Service (RADIUS) Implementation Issues and Suggested Fixes. RFC 5080, December 2007.

[35] The FreeRADIUS Server Project and Contributors. FreeRADIUS. https://freeradius.org/.

[36] Thomas H. Ptacek. RADIUS security. https://web.archive.org/web/20060819074129/http://skoda.sockpuppet.org/tqbf/radius-security.html, September 1996.

[37] Groupe Renault. Virtual private lte. https://www.telefonica.com/es/wp-content/uploads/sites/4/2021/10/Virtual-Private-LTE-Renault.pdf.

[38] Jan-Frederik Rieckers and Stefan Winter. (Datagram) Transport Layer Security ((D)TLS Encryption for RADIUS. Internet-Draft draft-ietf-radext-radiusdtls-bis-00, Internet Engineering Task Force, October 2023. Work in Progress.

[39] Carl Rigney. RADIUS Accounting. RFC 2866, June 2000.

[40] Ronald L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[41] Allan Rubens, Carl Rigney, Steve Willens, and William A. Simpson. Remote Authentication Dial In User Service (RADIUS). Internet-Draft draft-ietf-radius-radius-00, Internet Engineering Task Force, May 1995. Work in Progress.

[42] Allan Rubens, Carl Rigney, Steve Willens, and William A. Simpson. Remote Authentication Dial In User Service (RADIUS). RFC 2865, June 2000.

[43] Allan Rubens, Steve Willens, Carl Rigney, and William A. Simpson. Remote Authentication Dial In User Service (RADIUS). Internet-Draft draft-ietf-nasreq-radius-01, Internet Engineering Task Force, June 1994. Work in Progress.

[44] SITA. SITA wireless gatelink. https://www.sita.aero/globalassets/docs/use-cases/sita-wireless-gatelink-use-case.pdf.

[45] Spark. Private apn. https://www.spark.co.nz/iot/home/iot-solutions/private-access-point-name/.

[46] Marc Stevens. Project HashClash - MD5 and SHA-1 cryptanalytic toolbox, 2009. https://github.com/cr-marcstevens/hashclash.

[47] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.

[48] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and

Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2009.

[49] Evan Sultanik and Evan Teran. This PDF is an NES ROM that prints its own MD5 hash! In Manul Laphroaig, Melilot, Evan Sultanik, Jacob Torrey, Ange Albertini, Philippe Teuwen, and sundry others, editors, *PoC‖GTFO*, chapter 12, pages 56–59. Tract Association of PoC‖GTFO and Friends, 2017.

[50] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptol.*, 12(1):1–28, 1999.

[51] Heikki Vatiainen. Personal communication.

[52] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[53] Klaas Wierenga, Mike McCauley, Stefan Winter, and Stig Venaas. Transport Layer Security (TLS) Encryption for RADIUS. RFC 6614, May 2012.

[54] Ward Willats, Carl Rigney, and Pat R. Calhoun. RADIUS Extensions. RFC 2869, June 2000.

[55] Shalanda D. Young. Moving the U.S. government toward zero trust cybersecurity principles. https://www.whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf, January 2022.

[56] Kim Zetter. NSA laughs at PCs, prefers hacking routers and switches. https://www.wired.com/2013/09/nsa-router-hacking/, September 2013.

A On the origin of Proxy-State

The Proxy-State attribute was proposed for addition in an email to the IETF RADIUS working group on May 24, 1995 [3]. The justification provided in the email is to allow stateless proxying between clients and a pool of authentication servers. This attribute was included in the May 1995 protocol draft [41]. The only way that this attribute appears to be used in modern implementations is for catching proxy loops [16]. RFC 2865 mentions in at least seven different places that Proxy-State attributes must be returned in order and unmodified in responses. The presence of this attribute makes the protocol vulnerability much simpler to exploit than it would have been otherwise.

B Changes to hashclash

The code is available at [46]. We do not plan to publish our cluster scripts; they are uninformatively tailored to our setup. The following is an exhaustive summary of changes to `hashclash` that we made, categorized by program and goal.

`md5_birthdaysearch`:
- (Scaling) Add parameter to control how frequently trails are distributed/saved to and loaded by controllers.
- (Scaling) Add generatormode that only generates trails and distributes them to controllers.
- (Latency) Immediately save birthday collision when found, instead of waiting until all threads finish.
- (Fix) AVX256 allocation: Older GCC might not properly do large alignment as required for AVX256.

`md5_diffpathhelper`:
- (Section 4.1) Fix a few bytes at the start of a near-collision block.
- (Section 4.2) Add support to combine diffpaths, in order to use precomputed set of upper differential paths and overwrite the full differential paths with the correct ending differences for the near-collision block at hand.
- (Section 4.2) Add support to negate a set of diffpaths, in order to negate each 'positive delta m11'-based precomputed set of upper differential paths instead also computing the 'negate delta m11'-based set.
- (Fix) Add support to join files consisting of just a differential path (not a set of one).

`md5_diffpathforward`:
- Speed up differential path fast solvability checking: only check change, assume input paths were solvable.
- (Speed) Progressively increase output set size for the first few steps (t=1,2,3) of forward.
- (Speed) When full decrease maxcond to stop processing paths that will be pruned later on.
- (Speed) Use C++ move instead of copy.
- (Scaling) Use threadlocal buffers to global container to reduce contention.
- (Scaling, Latency) Add splitsave parameter to save output over multiple files in parallel.

`md5_diffpathconnect`:
- (Speed) New mintunnel parameter to prune search from start to avoid bad full paths with too few tunnels.
- (Latency) Parallel read of both inputfiles, new option to wait for file to exist before trying to load it.

All:
- (Scaling) Improve input distribution over threads.
- (Latency) Diffpath archives: gzip with `best_speed`.
- (Fix) Clean up comparison operators.
- (Fix) Make `dostep_index` volatile, read at start, write after critical section.