Reverse Engineering of Strong Crypto Signatures Schemes		
Data	by Evilcry	
05/11/2006	***UIC's Home Page***	**Published by Quequero**
Where ones sees a limit	Qualche mio...eventuale commento sul tutorial :)))	the others sees an opporunity
Prepare for what you cannot see, expect the unexpected, a waiting game waiting to see......	**WebSite:** **http://evilcry.altervista.org** **E-mail:** evilcry (GdoG) gmail (GdoG) com (To jump new catching techniques) **IRC frequentato** **irc.azzurranet.org #crack-it on efnet #RET**	Le radici profonde non gelano mai - Tolkien
Difficoltà	**NewBies** () **Intermedio** (**X**) **Avanzato** (**X**) **Master** ()	

Reverse Engineering of Strong Crypto Signatures Schemes
Written by **Evilcry**.

Introduzione

Ed eccomi qui a scrivere per il NYP2k7 (che più entropicamente contratto non si può), lo stile del paper sarà il classico approccio CryptoRev. Stavolta andremo a trattare principalmente con le ECC ovvero Elliptic Curve Cryptography, con particolare attenzione alle più comuni applicazioni che si trovano nel campo della sicurezza software, ed anche in parte hardware.

Tools usati

- Ida
- Ollydbg
- Buone basi di crittografia e matematica

Indice

1. Basic Intro
2. Introduction To Elliptic Curve Cryptography
3. Basic Math Background - Group Theory
4. Basic Math Background - Finite Field Arithmetic
5. Generalized Discrete Logarithm Problem
6. What Elliptic Curves are
7. Elliptic Curves in Cryptography
8. Elliptic Curve Arithmetic
9. Pratical Elliptic Curves Operations Sample
10. Foundamental Features of EC, and pratical ECC generation
11. Elliptic Curves Cryptographic Applications
12. Basic ECC Architecture Considerations
13. The Elliptic Curve Discrete Logarithm Problem and ECC Attacks
14. The Elliptic Curve Digital Signature Algorithm
15. A Reverse Engineering Approach

16. ECDLP and Schoof Solving, for Security Patterns

Notizie sul programma

KeygenMe 10 by WiteG -> Come target di un caso reale

Essay

Basic Intro

Perchè un Reverser dovrebbe studiare la crittografia?..alcuni erroneamente tendono a considerare le due discipline separate ed a sè stanti, ma lasciatevi dire da uno che abbastanza spesso si guadagna la pagnotta con robe del genere, qualcosa in merito..

Oggi moltissime protezioni Professionali basano la loro sicurezza su algoritmi crittografici più o meno forti, (dipende dal campo in cui ci si trova ad operare) e molte delle ricerche su future tecniche di protezioni, moltissime delle quali Hardware, tendono ad organizzare interi Schemi di Protezione, come **Sistemi Complessi** (maiuscolo non casuale) di algoritmi crittografici, spesso tenuti assieme da altrettanti algoritmi puramente matematici, che necessitano di **Attacchi Algebrici** comunque riconducibili alla metodologia crittoanalitica.

Attualmente le ricerche vertono principalmente sulla ricerca Implementativa su software, affiancato molto spesso da dispositivi hardware di varia natura, tra cui Chip, Smart Card in genere e dispositivi USB di Sicurezza, scelti in ovvia conseguenza all'importanza dei dati trattati.

Nel precedente elenco, c'è da fare una distinzione estremamente importante fra l'hardware usato, necessaria a dare una categorizzazione che da sola renderà l'idea del grado di sicurezza raggiungibile. I metodi comuni per la computazione efficiente di un algortimo possono essere in due grandi categorie:

Dispositivi **ASIC**: ovvero **Application Specific Integrated Circuit**, i quali sono specificamente richiesti in applicazioni di **Massima Sicurezza**, poichè oltre la grande Efficienza, hanno l'assoluto vantaggio di non essere modificabili dopo la produzione.

Dispositivi **FPGA**: ovvero **Field Programmable Gate Arrays**, contengono un array di elementi computazionali la cui funzionalità si ottiene attraverso un set specifico di istruzioni. Questi elementi si chiamano blocchi logici, e sono connessi assieme attraverso un set di *routing resurces* anch'esse programmabili.

Ultimamente le analisi sulla sicurezza degli FPGA, hanno fatto la loro comparsa su internet alla grande massa, come *Techniques of FPGA Exploitation*, anche se per (s)fortuna rimane un campo per il momento precluso a molti.

Arrivando rapidamente al punto, per noi, di maggiore interesse, c'è da dire che gli algoritmi implementati più diffusi sono AES, RSA ed ECC. Quella maggiormente interessante a mio parere è l'ultima, ovvero la ECC, poichè offre un grado di sicurezza equivalente ad RSA con key size molto piccole, e ciò diviene ancora più vantaggioso nell'hardware, dato che offre Alta Velocità di esecuzione, Bassi Consumi e piccoli Certificati. Quest'ultima cosa è di fondamentale importanza, nell'hardware più diffuso (Smart Card, Telefonia Mobile, PDA).

Quindi, conoscendo approfonditamente le possibilità di analisi e Reverse su software, anche se a livello implementativamente diverso, abbiamo lo stesso metodo di analisi necessario per lavorare anche sull'hardware.

Introduction To Elliptic Curve Cryptography

I crittosistemi a chiave pubblica (PKC), più diffusi ed utilizzati sono quelli basati sulla fattorizzazione (RSA) e sull'uso del Logaritmo Discreto (Diffie-Hellman, ElGamal, Schnorr, DES). Questi permettono una comunicazione sicura, su canali fondamentalmente insicuri. L'alternativa che ormai sta diventando una Main Stream è la **Crittografia basata sulle Curve Ellittiche** o più comunemente **ECC**, questo per il grande numero di vantaggi e possibilità che può offrire, in una parola possiamo definirla *estremamente flessibile*. Le Curve Ellittiche proposte, possono infatti essere adottate nella PKC basata su fattorizzazione sia su quella basata sul problema del logaritmo discreto o **DLP**. E' fondamentale però, parlare delle grandi differenze che sussistono fra l'uso dei due tipi di sistemi (fattorizzazione o DLP); le ECC basate sul primo tipo, sono essenzialmente di uso accademico poichè non offrono vantaggi differenti dal RSA (eccetto l'uso di KeySize più piccoli), e data la loro sicurezza sono applicati alla stessa gamma di problemi risolvibili da RSA.

Discorso diverso meritano le ECC basate sul DLP, poichè la sicurezza di queste "Varianti" dipendono strettamente sulla (ridefinizione, nuova importanza concettuale assunta da..) degli algoritmi usati per la risoluzione dei classici DLP. Questo nuovo aspetto (ricordo che mi sto definendo agli algoritmi classici) ridefinisce la normale *Esponenziazione*, in **Sub-Exponentiation Time**, se applicati alla *Risoluzione delle Curve Ellittiche*. Se invece ci riferiamo ad algoritmi più generali, progettati specificatamente per ECC, si parla di **Tempo Esponenziale**. In termini più sintetici, aspetti dello stesso algoritmo assumono termini diversi, a seconda dell'efficacia che hanno su DLP o **ECDLP**.

Tutto iniziò nel 1984 grazie ad un grande crittografo e matematico, Hendrik Lenstra, il quale descrisse un algoritmo di fattorizzazione estremamente interessante e funzionale basato sulle proprietà delle Curve Ellittiche, chiamato appunto Lenstra Elliptic Curve Factorization, che però non possiamo ancora definire come ECC, poichè è un'applicazione matematica non crittografica. Le vere ECC, nacquero un anno dopo (1985) ad opera di **Neal Koblitz** e **Victor Miller**, i quali si basarono su un presupposto fondamentale, che fu quello di (riadattare/ridisegnare) la crittografia a chiave pubblica, su un'impalcatura matematica che basa le proprie caratteristiche peculiari sulla Struttura Algebrica fornita dalle **Curve Ellittiche** su **Campi Finiti**.

Basical Math Background - Group Theory

Per rendere più completo e maggiormente accessibile l'articolo, ho inserito questo piccolo paragrafo, riguardo il background matematico necessario, con particolare attenzione alla **Teoria dei Gruppi**.

Un *Gruppo Abeliano* $(G,*)$ consiste in un insieme *G* con un'operazione binaria del tipo

$$* = G \times G \rightarrow G$$

che soddisfa le seguenti proprietà:

- **Associatività** $a * (b * c) = (a * b) c$ per tutte la *a, b,c* appartententi a *G*
- **Elemento Identità**, esiste un elementi identità *e* appartente a *G* tale che $a * e = e * a = a$ per tutte le *a* appartenenti a *G*

- **Esistenza degli Inversi**, Per ogni a appartente a G esiste un elemento b chiamato *Inverso* di a, tale che $a * b = b * a = e$
- **Commutatività**, $a * b = b * a$

Possiamo fondamentalmente avere due tipi di gruppi, quelli *Additiv*i (+) e quelli *Moltiplicativ*i (*). Questa distinzione deriva dal fatto che un gruppo è detto additivo quando l'elemento identità a è zero, mentre l'inverso è $-a$. I gruppi moltiplicativi sono così definiti qualora l'elemento identità sia 1 e l'inverso indicato come $a^{(-1)}$. Infine un gruppo è chiamato **Finito**, quando G è un insieme finito, in questo caso abbiamo anche un *Ordine*, che in pratica definisce il numero di elementi di G.

Per esempio, fissato un numero primo p, possiamo facilmente costruire un gruppo finito di ordine p, $\mathbb{F}_p = \{0,1,2,...,p-1\}$ dato appunto da un insieme di interi. Inoltre per quanto visto prima possiamo considerare per questo gruppo $(Fp, +)$ ovvero un gruppo Additivo in modulo intero p con elemento identità 0, ed anche un gruppo $(\mathbb{F}_p*, *)$, dove $\mathbb{F}_p*$ indica gli elementi *non-nulli* dell'insieme da noi considerato, e costituisce inoltre (com'è ovvio) un gruppo *Moltiplicativo* di ordine $p-1$ avente come elemento identità 1.

A questo punto potreste chiedervi, esiste un gruppo unico che comprende allo stesso momento l'operazione Additiva e quella Moltiplicativa? così evitiamo di portarci dietro due bagagli?..la risposta è sì, sta tutto nella tripletta $(\mathbb{F}_p,+,*)$ ovvero i **Campi Finiti**. Alla luce di quanto detto, possiamo definire G come un Gruppo Moltiplicativo Finito di ordine n, e possiamo introdurre ulteriori elementi caratteristici dei campi finiti, come l'elemento g appartenente G, con il più piccolo intero positivo dato da t, definito come:

$$g^t = 1$$

chiamato *Ordine di g*, la cui diretta e più importante conseguenza è quella di esistere sempre ed essere un divisore di n. Una proprietà molto interessante del concetto fondamentale di gruppo, è l'effetto matrioska che si genera, possiamo infatti definire un insieme:

$$<g> = \{g^{(i)} : 0 <= i <= t - 1\}$$

ovvero l'insieme di tutte le potenze di g, il quale come avrete capito costituisce già "di per se" un gruppo interno (sottogruppo) a G, ed è chiamato appunto *Sottogruppo Ciclico* di G generato da g. Per la nozione di Campo Finito, quanto detto finora vale anche per G scritto con regole additive, più precisamente l'ordine di g è il più piccolo divisore positivo t di n, o ancora meglio:

$$t * q = 0$$

e di conseguenza:

$$<g> = \{ i * g : 0 <= i <= t - 1 \}$$

Nella nozione Additiva $t*g$ assume il significato elemento ottenuto aggiungendo t copie di g. Possiamo infine sintetizzare quanto dicendo che *se G possiede un elemento g di ordine n, allora G è detto Gruppo Ciclico e g verrà definito come Generatore di G.*

Finite Fields Arithmetic

L'**Aritmetica dei Campi Finiti**, è la base fondamentale di ogni sistema che utilizzi le proprietà delle Curve Ellittiche, principalmente in crittografia, la corretta implementazione degli algoritmi sui Campi Finiti, è il primo più importante presupposto che determina l'efficienza e la sicurezza di un sistema ECC. Esistono principalmente tre generi di campi finiti e sono i **Campi Primi** (ovvero formati da numeri primi), **Campi Binari** e le **Estensioni Ottimali dei Campi**, per ognuno di queste categorie, esistono difficoltà implementative crescenti nell'ordine in cui li ho elencati, è inoltre necessario adottare una diversa gamma di algoritmi per ogni genere di Campo.

I diversi algoritmi, se pur con diversi nelle implementazioni, seguono un unico concetto fondamentale, che è quello di *Eseguire Operazioni Aritmetiche, NEI campi e FRA i campi*, ovvero lavorando come *Connettori (Mixer)* oppure come *Singoli Operatori*.

I **Campi**, sono astrazioni o meglio sottoinsiemi chiamati F dei vari sistemi di numerazione che conosciamo. Ammettono principalmente due operazioni, *Addizione* e *Moltiplicazione*, e possiedono esattamente le stesse proprietà viste per i Gruppi.

In realtà, le operazioni possibili con i campi sono quattro, ovvero si aggiungono anche *Sottrazion*e e *Divisione*, come tipi derivati dalle due principali. La *Sottrazione* è definita in termini additivi come: *a - b = a + (-b)*, mentre la *Divisione* è definita in termini di moltiplicazione, come: *a/b= a * b ^ (-1)* con $b \neq 0$e *b ^ (-1)* è l' elemento inverso, ovvero quell'elemento che rispetta la proprietà *b * b ^ (-1)=1*.

Campi Finiti Primi:I campi finiti primi si indicano con $\mathbb{F}_p$, dove p è un numero primo maggiore di 3, definito anche come modulo di $\mathbb{F}_p$. Per ogni intero *a*, (*a MOD p*) avremo un unico resto *r* compreso sempre fra *0* e *p*, l'operazione necessaria per trovare *r* viene invece definita come **Riduzione Modulare**.

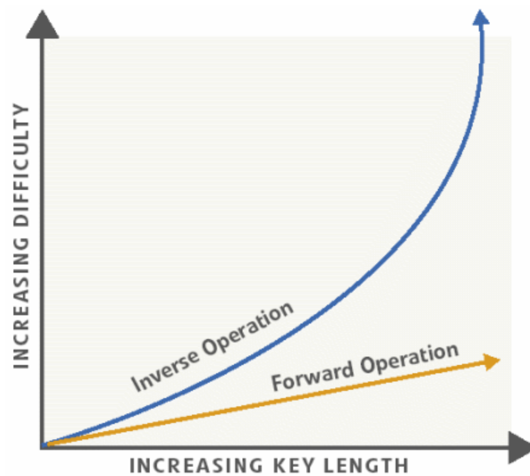
Considerato ad esempio un campo F29, i suoi elementi saranno ovviamente:

F29={0,1,2,...,28}

ed in virtù delle proprietà dei campi potremo definire alcune operazioni artmetiche di base:

- 17+28 = 8 ovvero 37 MOD 29 = 8 **(Addizione)**
- 17-20 = 26 ovvero -3 MOD 29 = 26 **(Sottrazione)**
- 17 * 20 = 21 ovvero 340 MOD 29 = 21 **(Moltiplicazione)**
- 17^ (-1) = 12 ovvero (17*12) MOD 29 = 1 **(Inversione)**

Con questo piccolo esempio, oltre a capire come funzionano le operazioni aritmetiche su campi finiti, possiamo giungere ad un'altra fondamentale osservazione, **l'utilizzo dei campi finiti è il principale strumento per la riduzione della complessità computazionale di un algoritmo in termini di efficienza**, infatti come potete notare una semplice somma 17+28 = 8, a voi le conclusioni sulle enormi potenzialità di adottare i campi finiti in crittografia :)



Quello che vedete qui è il grafico di un generico algoritmo **Asimmetrico** (ovvero il caso delle curve ellittiche), potete inoltre notare che dato un key-length *x* sufficientemente lungo da poter dire che l'operazione di Inversione risulta così difficile che il sistema può essere definito sicuro, e contemporaneamente che *x* non è così lunga che il normale utilizzo (forward operation)

Campi Binari: I campi binari si indicano con $\mathbb{F}_{2^m}$ chiamati anche **campi finiti caratteristici di ordine 2**. Possono essere considerati come uno spazio vettoriale *m* in F2, definito dagli elementi *0-1*. Dalle nozioni base di algebra lineare (che dovreste avere, altrimenti meglio che (leggiate qualcosa in merito, almeno fino agli Spazi Vettoriali basi e dimensioni) esistono *m* elementi *a*, definibili con una combinazione di vettori linearmente indipendenti che formano una base (*m-1*). Alla stregua della più classica Algebra lineare, tratteremo questo insieme come una bitstring, scoprendo automaticamente che l'operazione di Addizione si ottiene attraverso un XOR tra due vettori di bits, la moltiplicazione invece è direttamente correlata alla base scelta. Le basi utilizzabili in $\mathbb{F}_{2^m}$ sono numerose, ma l'utilizzo a scopi computazionali è stato ristretto a poche classi, poichè è stato scoperto che alcune classi sono più efficienti rispetto ad altre. La scelta può comunque oscillare fra **basi polinomiali** e **basi normali**, noi tratteremo escluivamente con la **Rappresentazione in Base Polinomiale**. Viene scelto un polinomio irriducibile *f(z)* di grado *m*, con l'irriducibilità intendiamo dire che **f(z)** non può essere fattorizzato come prodotto di più polinomi base di grado inferiore ad m. Adiamo ora a definire le proprietà ed i meccanismi delle operazioni aritmetiche in $\mathbb{F}_{2^m}$ con base polinomiale alla maniera precendente vista.

Considerato un campo binario F2^4, gli elementi ad esso appartenenti sono 16 polinomi binari di grado massimo 3.

0	*z^2*	*z^3*	*z^3+z^2*
1	*z^2+1*	*z^3+1*	*z^3+z^2+1*
z	*z^2+z*	*z^3+z*	*z^3+z^2+z*
z+1	*z^2+z+1*	*z^3+z+1*	*z^3+z^2+z+1*

Passiamo ora alle possibili operazioni base:

- *(z^3+z^2+1)* + (z^2+z+1) = *z^3+z*
- *(z^3+z^2+1)* - (z^2+z+1) = *z^3+z* (da tener presente che in F2 (-1 = 1))

- *(z^3+z^2+1)* * (z^2+z+1) = *z^2+1* **ovvero** *(z^3+z^2+1)* * (z^2+z+1) = z^5 + z + 1 **SEGUE** (z^5 + z + 1) MOD (z^4+z+1) = *z^2+1*
- *(z^3+z^2+1)*^(-1) = z^2 ovvero *(z^3+z^2+1)* * z^2 MOD (z^4+z+1) = 1

Avrete certamente capito che il secondo membro del MOD (z^4+z+1) non corrisponde ad altro che al polinomio irriducibile f(z)=z^4+z+1

Generalized Discrete Logarithm Problem

Andando adesso a considerare ad analizzare il DL, potremo vedere un'applicazione pratica della Teoria dei Gruppi, e potremo trovarne una pratica concretezza. In qualunque sistema basato sul logaritmo discreto possiamo trovare un insieme di parametri di pubblico dominio, ovvero *(p, g, q)*, dove *p* è un qualunque numero primo, *q* è un divisore (anch'esso primo) di *p-1* e *q* Appartiene all'intervallo *[1, p-1*], ed ha Ordine *q*, in altre parole ciò che abbiamo appena visto, *t = q* il più piccolo valore che soddisfa la relazione:

$$g^t = 1 \pmod p$$

Ora dovreste cominciare a considerare quest'algoritmo con occhi leggermente diversi :), infatti con qualche piccola assunzione possiamo ridefinire l'intero processo di encryption, del Logaritmo Discreto. Supponendo infatti che *(G,*)* sia un gruppo ciclico moltiplicativo di ordine n ed avente come generatore *g*, possiamo "includere" l'intero algoritmo DL, all'interno dello stesso *G*!. Considerando infatti che i parametri di dominio sono *g* ed *n*, e la chiave privata è un intero *x*, selezionato a caso, nell'intervallo *[1,n-1]* e la chiave pubblica sia di conseguenza data da:

$$y= g^x$$

Il problema di determinare *x* dati *g*, *n* ed *y* è definito **Problema del Logaritmo Discreto** (DLP) in *G*, c'è da dire inoltre che per un sistema DL basato su *G* dovrebbe essere intrattabile, ma esiste fondamentalmente un particolare aspetto che riduce l'efficienza computazionale e di sicurezza (ovviamente avrete capito, che il discorso vale sia per l'attacco che per l'efficienza stessa dell'algoritmo) del DL. Quest'ultimo discorso sarà molto più chiaro, specificando una particolare proprietà dei gruppi, ovvero che ogni due Gruppi Ciclici dello stesso ordine *n*, questi(i gruppi stessi) possono essere considerati operativamente simili, più semplicemente sono come due contenitori uguali ma con un diverso contenuto, o (seconda) possibilità, sarebbe possibile scrivere gli elementi del secondo gruppo con una forma differente. L'immediata conseguenza computazionale che otteniamo, è quella che a differenti rappresentazioni, corrispondono altrettante differenti curve di efficienza (velocità), sia che si tratti del DL sia del DLP.

Potreste chiedervi, cosa accomuna il DL con le curve ellittiche, le relazioni sono tante, in primis le curve ellittiche sono dirette applicazioni nel campo dei Finite Fields, mentre le DL possono essere interpretate in termini di Finite Fields, questo ha portato i ricercatori ad unire le proprietà curve ellittiche a quelle dei DL.

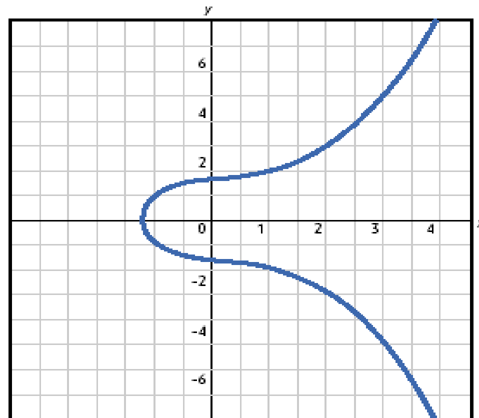
What Elliptic Curves are

Una Curva Ellittica, è una curva piana determinata da un'equazione del tipo:

$$y^2 = x^3 + ax + b$$

La principale caratteristica (agli scopi crittografici) di questo genere di curve, è principalmente quella che l'insieme dei punti della curva stessa forma un **Gruppo Abeliano** avente come elemento Identità un punto all'Infinito. Se le coordinate *x* ed *y* sono estratte da un **Campo Finito** sufficientemente grande, l'insieme delle loro soluzioni andrà a formare un Gruppo Abeliano Finito. Se ben ricordate quanto ho scritto nel paragrafo sul background matematico, il DL può essere assimilato come un insieme di gruppi ciclici finiti, potrete ben capire che il passo da gruppo a punti di una curva ellittica (date le precedenti assunzioni) è relativamente breve, ma con un fondamentale **Aumento di Complessità**, che è proprio il punto di forza delle ECC.

Eccovi infine il grafico di una generica curva ellittica ottenuta da *y^2 = x^3 + ax + b*:



Elliptic Curves in Cryptography

Nella sua accezione fondamentale la Curva Ellittica non è che il **MEZZO**, poichè le proprietà algebriche della sua forma geometrica sono utilizzate per definire gli elementi dell'insieme da cui si calcola il Gruppo. Considerate un grafico contenuto in un riquadro p x p, dove p è il nostro solito numero primo, ovviamente il campo $\mathbb{F}_p$che ne otterremo andrà da *0* a *p-1*; le operazioni algoritmiche coinvolgeranno quindi i punti di questa curva che dovranno quindi rispettare la condizione di appartenere ad $\mathbb{F}_p$. Le Curve Ellittiche usate in crittografia possono essere di **due classi** di appartenenza ai Campi Finiti, la **prima riguarda** $\mathbb{F}_p$(con *p>3*) mentre la **seconda** riguarda $\mathbb{F}_{2^m}$, nel caso invece di una generica applicazione sui Campi Finiti, si utilizza $\mathbb{F}_q$(che in altre parole è un Campo Esteso Ottimale) dove *q = p*.In $\mathbb{F}_p$gli elementi sono essenzialmente interi $0 \leq x < p$ risultanti da operazioni di Aritmetica Modulare. Le curve basate invece su $\mathbb{F}_{2^m}$ sono quelle più complesse, dato il maggior numero di possibili rappresentazioni (qui vale lo stesso discorso per l'efficienza dei DL o DLP) come bitstring per ogni polinomio binario irriducibile (*f(z)*) di grado *m*.

L'insieme delle coppie di coordinate affini (x,y) con $x, y \in \mathbb{F}_q$ formano il Piano Affine $\mathbb{F}_q \times \mathbb{F}_q$, date queste premesse possiamo ricavare direttamente la definizione di Curva Ellittica (che chiameremo E).

Una Curva Ellittica E, è il luogo geometrico dei punti nel piano affine, le quali coordinate soddisfano l'equazione $4a^3 + 27b^2 \neq 0$(*MOD p*), avente come punto all'infinito *O*, ovvero il punto dove il luogo geometrico nel piano proiettivo interseca la linea all'infinito. Nel caso più

semplice (ovvero *p > 3*) troveremo che la $E(\mathbb{F}_p)$ (il quale indica l'insieme di tutti i punti su E) si traduce nella forma che abbiamo già visto, ovvero *y^2 = x^3 + ax + b*, dove a e b appartengono ad $\mathbb{F}_p$. Il discorso potrebbe risultare alquanto astratto, se non siete molto ferrati in Algebra, facciamo quindi un esempio molto semplice riguardante la classe di curve ellittiche più semplice ovvero le $\mathbb{F}_p$; per esempio, se E è una curva ellittica in F7 avente quindi come *equazione di definizione*: *y^2 = x^3 + ax + b*, allora i punti saranno:

E(F7) = {Infinity, (0,2), (0,5), (1,0), (2,3), (2,4), (3,3), (3,4), (6,1), (6,6) }

Passiamo ora alle $E(\mathbb{F}_{2^m})$ la cui *equazione di definizione* può essere scritta come: *y2 + xy = x3 + ax2 + b* dove *a* e *b* appartenenti ad $\mathbb{F}_{2^m}$ sono costanti, inoltre $b \neq 0$ per *O*=(0,0) e negli altri casi *O*=(0,1); in base poi al **Teorema di Hasse** sulle curve ellittiche possiamo quantificare il numero di punti di una curva utilizzando la seguente diseguaglianza:
$(\sqrt{q}-1)^2 \leq |E(\mathbb{F}_q)| \leq (\sqrt{q}+1)^2$.

Elliptic Curve Arithmetic

Ci concentreremo adesso sui Punti di una Curva Ellittica, poichè tutti i meccanismi crittografici legati alle ECC, si basano sull'**Aritmetica dei Punti** che diverrà per noi lo sarà lo strumento più importante con cui andremo ad operare in fase di progetto, analisi e reverse. I punti di una curva ellittica, come già detto formano un gruppo abeliano $(E(\mathbb{F}), +)$ avente come al solito *O* come punto distinto all'infinito, ed a cui è assegnato il ruolo di **Identità Additiva**, ovvero considerati due punti $P, Q \in E(\mathbb{F}_q)$, avremo un Terzo punto, definito come *P*+*Q* su $E(\mathbb{F}_q)$, ottenendo di conseguenza $P, Q, R \in E(\mathbb{F}_q)$.

Essendo i punti, gli elementi che costituiscono un gruppo abeliano, per loro valgono le stesse regole viste precedentemente per le operazioni in *G*, ma che conviene adesso riscrivere per mettere in evidenza l'importanza del punto *O*:

- *P + Q = Q + P*
- *(P+Q) + R = P + (Q + R)*
- *P + O = O + P = P*
- *-P* così che *-P + P = P + (- P)= O*

Alla luce di queste proprietà, possiamo finalmente introdurre le Operazioni Fondamentali su Curva Ellittica, vi consiglio inoltre di porre molta attenzione in questa parte, dato che la quasi totalità degli algoritmi lavorano essenzialmente sulle Operazioni su EC.

Addizione di una Curva Ellittica: Procederemo con un approccio geometrico ed uno algebrico, per meglio comprendere il reale significato di addizione di una EC. Esiste una regola chiamata **Corda e Tangente** che ci permette di sommare due punti di una curva ellittica definiti come $P, Q \in E(\mathbb{F}_q)$, dalla somma otteniamo un terzo punto

P + Q = R

R è di conseguenza un punto appartenente alla curva ellittica stessa, possiamo inoltre vedere (consultando la tabella delle proprietà dei Gruppi Abeliani, appena citata) come definendo il

negativo di un punto *P = (x,y)* in virtù delle proprietà appena viste avremo - *P = (x, - y)* per $P \in E(\mathbb{F}_p)$ e - *P = (x,x + y)* per $E(\mathbb{F}_{2^m})$, possiamo quindi definire le seguenti regole per l'addizione:

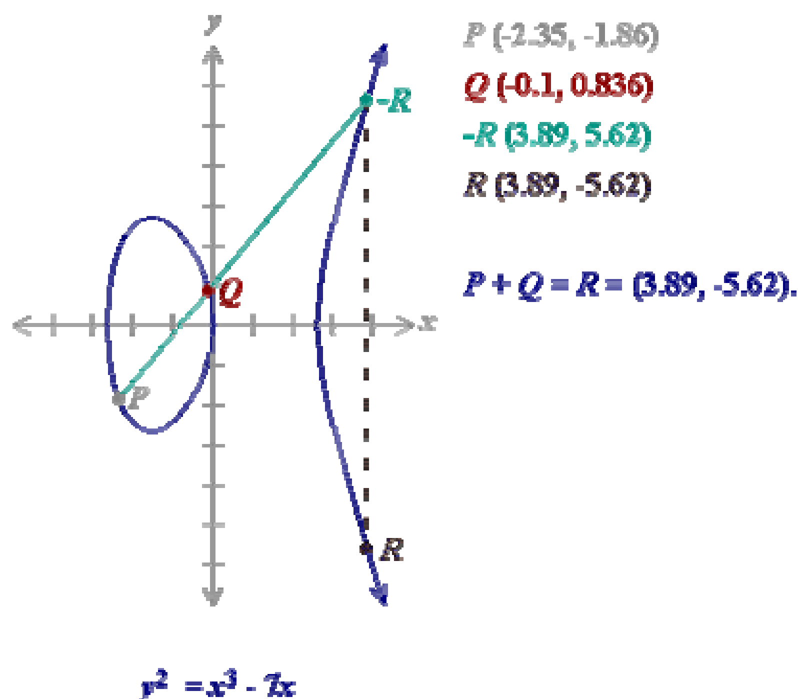
- se *Q = O* allora *P + Q = P*
- se *Q = - P* allora *P + Q = O*
- se $Q \neq P$ allora *P + Q = R*

Per quest'ultimo caso è necessario distinguere fra:

- **Campi Primi**: *xR = l^2 - xP - xQ, yR = l(xP - xR) - yP* dove *l* = $\lambda = \frac{y_Q - y_P}{x_Q - x_P}$
- **Campi Binari**: *xR = l^2 + l + xP + xQ + a, yR = l(xP + xR) + xR + yP* dove *l* = $\lambda = \frac{y_P + y_Q}{x_P + x_Q}$

Vedete come algebricamente $E(\mathbb{F}_q)$ andrà a formare un Gruppo con *O* elemento identità, otterremo così la ECC vera e propria!

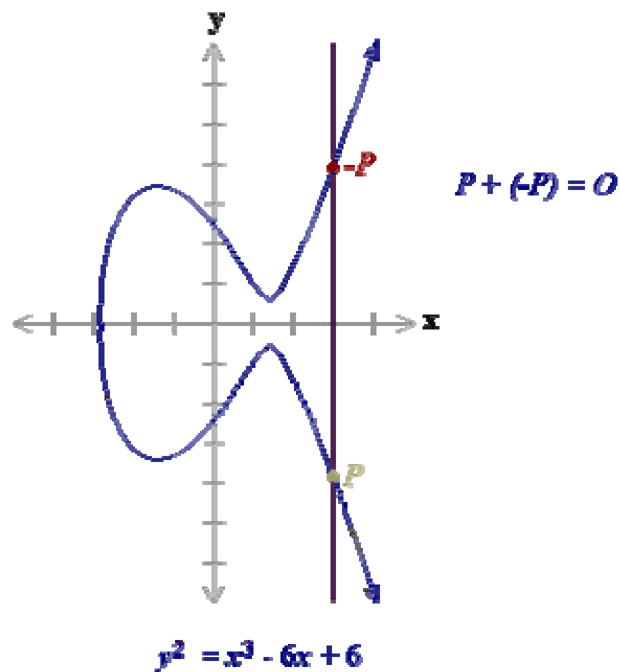
Passiamo adesso all'**Approccio Geometrico**. Consideriamo *P* e *Q* due punti appartenenti alla curva ellittica E, tracciamo la retta congiungente che passa fra *P* e *Q* e prosegue fino ad intersecare la curva ellittica, ovviamente individuerà un punto, che Riflesso lungo l'asse *x* andrà ad individuare il nostro punto *R(x3,y3)*!



Come avrete certamente capito, il caso appena citato riguarda solo uno dei possibili casi di addizione passiamo al caso:

P - P* Addiction**: ***P + (-P) = O

La congiungente *P -P* è una retta verticale che ovviamente non "genera" un terzo punto *R* e perciò si estende all'infinito individuando il punto *O*, per induzione capirete inoltre che ***P + O = P***

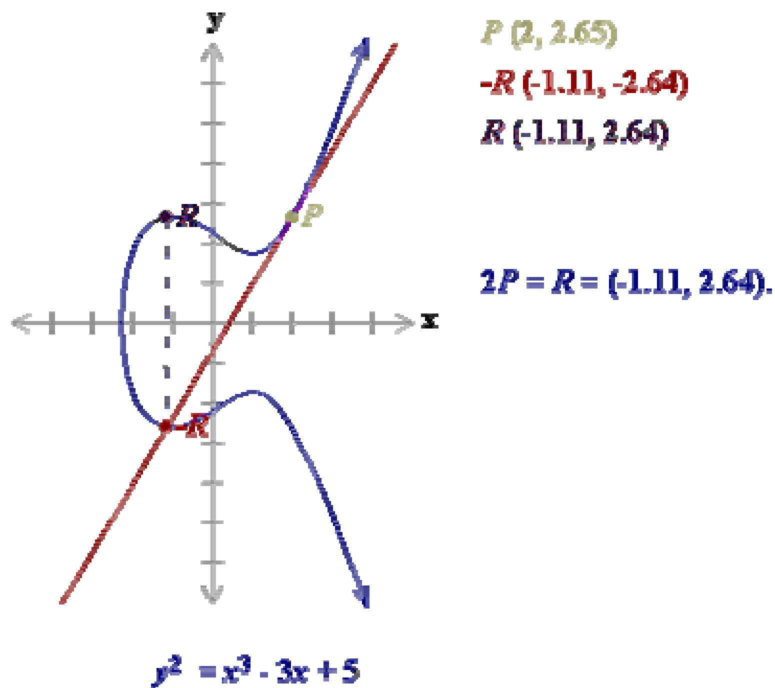


Con quest'ultimo caso abbiamo concluso la trattazione dell'Addizioni, che comunque ritroveremo inseguito applicata direttamente ad un algoritmo crittografico.

Duplicazione di *P, 2P*: Possiamo esplicare la duplicazione di *P*, come:

2P = P + P = R

Dal punto di vista geometrico questo equivale alla seguente costruzione, si disegna prima la tangente alla curva nel punto *P*, il suo prolungamento individuerà un punto, che riflesso lungo l'asse *x* ci darà *R*:

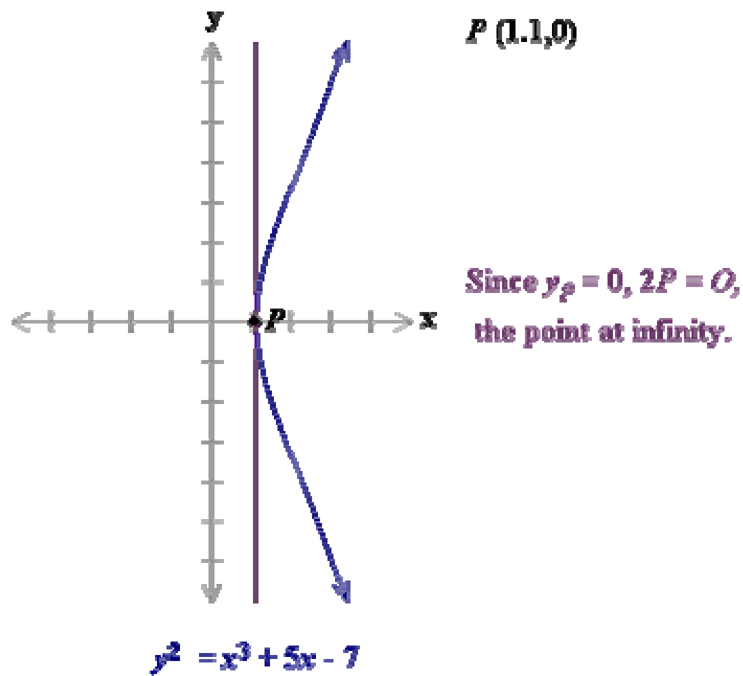


Torniamo adesso un attimo a considerazioni puramente analitiche, possiamo individuare due casi algebrici legati all'operazione di duplicazione:

- **Campo Finito Primo**: *xR = l^2 - 2xP, yR = l(xP - xR) - yP* dove *l* = $\lambda = \frac{3x_P^2 + a}{2y_P}$
- **Campo Binario**: *xR = l^2 + l+ a, yR = xP^2 + (l + 1) x_R* dove *l* = $\lambda = x_P + \frac{y_P}{x_P}$

Duplicazione di *P*, se yP = 0: La tangente da *P* è sempre verticale se la componente yP = 0, di conseguenza se andiamo a duplicare un punto con *yP = 0*, essendo quest'ultima una tangente alla curva ellittica stessa non intercetterà mai un punto *R*, tendendo quindi all'infinito, ovvero all'Identità Additiva *O*, per cui:

2P = O



Volendo trovare *3P* (sempre nel caso di *yP = 0*), avremmo avuto che *3P = 2P + P*; ovvero *P + O = P*, e quindi *3P = P*; per gli altri *4P = O*, *5P = P*, *6P = O* ecc.

Pratical Elliptic Curves Operation Samples

Con questo si completa il piccolo quandro riassuntivo sulle Operazioni su Curva Ellittica, ma prima di cambiare argomento vi riporto due esempi pratici di operazioni su EC su Campi Finiti Primi:

Elliptic Curve Addition: consideriamo il punto *P*(4,7) e *Q*(13,11), ***P + Q = (X3,Y3)*** sarà determinato come:

X3 = ((11 - 7) / (13 - 4))^ 2 - 4 - 13 =

= 3^2 - 4 - 13 = -8 =

= **15** MOD 23

Mentre per la coordinata *y*:

Y3 = 3(4-15) - 7 = -40 =

6 MOD 23

di conseguenza *R* = (15,6)

Elliptic Curve Point Doubling: consideriamo adesso il punto *P*(4,7), ***2P = (X3,Y3)*** sarà determinato come segue:

X3 = (confrontate la Lambda per il point doubling)

((3*4^2 + 1) / (14))^2 - 8 =

=15^2 - 8 =

= 217 =

= **10** MOD 23

Y3 = 15(4 - 10) - 7 =

= -97 =

=**18** MOD 23

Il nostro R, sarà quindi **R = (10, 18)**

Ah..spero non vi stiate chiedendo da dove viene quel mod 23 :), è ovvio che dipende dal campo scelto ovvero un *F23*!

Per quanto riguarda la computazione Additiva e Duplicativa su $E(\mathbb{F}_{2^m})$, il discorso rimane pressocchè invariato, l'unica differenza è che invece di avere numeri puri, tratteremo con dei polinomi irriducibili.

Foundamental Features of EC, and pratical ECC generation

Ordine del gruppo: Considerando la curva $E(\mathbb{F}_q)$, grazie al **Teorema di Hasse sulle Curve Ellittiche**, possiamo determinare il numero di punti di E incluso O, eccovi la relazione fondamentale di Hasse:

$$|\sharp E(\mathbb{F}_p) - p - 1| < 2\sqrt{p}$$

Computazionalmente ciò è riassunto dall'algoritmo di **Schoof** conosciuto anche come **SEA** (**Schoof-Elkies-Atkin**).

La conoscenza del numero di punti di E è di fondamentale importanza per la crittografia in termini di implementabilità e cosa fondamentale in termini di sicurezza..

Supponete infatti di sommare un *P+P+P...* sufficientemente grande, essendo nel campo dei numeri finiti, facilmente potremmo raggiungere il punto O, e questo non è affatto una bella cosa..ve lo dimostro..

ci troveremmo di fronte ad una situazione del tipo:

*a*P = b*P* per qualche *a, b* con *b > a* questo implica l'esistenza di ***c*P = O*** dove ***c = b - a*** !!!

ECC Generation

questa non è altro che l'ordine del gruppo e di conseguenza PUO' dividere l'ordine del gruppo stesso! Capite adesso l'importanza di generare ECC di ordine ben definito..per fare ciò possiamo utilizzare l'algoritmo di Schoof (non mi sono ripetuto, sopra è utilizzato indirettamente) oppure

algoritmi estremamente più complessi come ad esempio la **Moltiplicazione Complessa**, od il più immediato **Teorema di Weil**, se invece ci troviamo su dispositivi hardware i più diffusi sistema di **Point Counting** sono l'**AGM** (**Arithmetic Geometric Mean**) ed il **SST** (**Satoh-Skjernaa-Taguchi**) che risultano essere estremamente veloci e performanti sui campi binari.

Altra condizione a cui devono sottostare i punti fissati sono la **MOV Condition** (Menezes Okamoto e Vanstone), molto spesso infatti potrà accadervi anche nella pratica professionale di avere la possibilità di ridurre una complessa ECC in una forma logaritmica e vi assicuro, non è poco, anzi direi proprio vergognoso :)

MOV Condition: Supponiamo di avere una ECC su GF(q) ed un punto fissato F, per prima cosa si controlla che i primi T termini della sequenza $(q, q^2, q^3..., q^T)$ siano diversi da *1 mod Ordine_Primo di F*. Si sceglie di solito

$$T = log2(q)/8$$

Non sto a spiegarvi l'intera procedura di attacco, che potete comunque trovare facilmente su CACR, vi basti sapere che i **logaritmi della curva ellittica vengono ridotti a comuni logaritmi su campi finiti**!

Da notare inoltre che un campo di q elementi deve avere un ordine diverso da q, o ci troveremmo nel caso di **Curve Ellittiche Anomale**, delizia di chi attacca!

Point Compression: Se conosciamo la componente x di un punto, possiamo trovare la y risolvendo ovviamente l'equazione della curva, essendo un'equazione di tipo Quadratico però, ci servono ulteriori bits d'informazione per determinare quale fra le soluzioni è quella corretta.

Elliptic Curves Cryptographic Applications

Eccoci finalmente arrivati alla parte più pratica ed interessante dell'intero discorso, ovvero di queste curve ellittiche, cosa ne facciamo?..il buon crittografo ha due soluzioni:

- **Uso prettamente crittografico**
- **Fattorizzazione di Interi**
- **Primality Proving**

La fondamentale assunzione fatta in crittografia, è *come riadattare un certo algoritmo che utilizza la matematica dei gruppi, ai gruppi basati su curva ellittica.* Potremo infatti trovarci ad algoritmi come:

- **ECDSA**
- **ECDH Elliptic Curve Diffie-Hellman**
- **ECMQV basato su MQV**
- **ECIES**
- **EC-KDSA**

Basic ECC Architecture Considerations

Qualsiasi implementazione di curva ellittica applicata ad algoritmi crittografici possiede un'architettura di base solida e comune a tutti gli algoritmi; questa struttura prende comunemente il nome di **Dominio dei Parametri**, ma suona meglio in inglese come **Domain Parameters**.

Domain Parameters: Questa struttura definisce la curva ellittica E, in funzione del **Campo** di un **Punto Base G** e dell'**Ordine *n*** del campo.

Solitamente in Dominio si indica essenzialmente come: **D = (q,FR, S,a,b, P,n,h)**

- *q* è l'ordine del Campo scelto
- *FR* (Field Representation) indica la rappresentazione usata per gli elementi di $\mathbb{F}_q$
- *S* (Seed) nel caso la curva ellittica necessiti di un PRNG
- *a,b* i due coefficienti dell''Equazione di Definizione
- *P* o *G* è il **Punto Base** o **Generatore** (si tratta di un Sottogruppo Ciclico come dovreste ricordare), ai fini crittografici l'Ordine di *G* dev'essere il più piccolo numero primo n non negativo così che $nG = O$.
- *n* è l'ordine *n* del punto base
- *h* è chiamato **Cofattore**

Dato che è l'ordine di $E(\mathbb{F}_q)$, per il Teorema di Lagrange $h = \frac{|E|}{n}$, nelle applicazioni crittografiche il Cofattore dev'essere $h \leq 4$e solitamente si utilizza $h = 1$.

I Domain Parameters, sono gli elementi più delicati di una ECC ed i primi che un ipotetico attacker andrà a studiare è necessario infatti che rispettino una serie di norme descritte dal **NIST** e dal **SECG**; gli attacchi più classici sono il *Pohlig-Hellman* ed il *Pollard's rho*, strettamente dipendenti dal numero di punti della curva ellittica, non mi soffermo qui a parlare degli attacchi anche perchè sarebbe necessario un articolo a parte data la complessità dell'argomento.

Key Pairs: Le chiavi utilizzate nelle ECC sono strettamente legate ai parametri del dominio, la **Chiave Pubblica** è scelta casualmente selezionando un punto Q del gruppo <*P*> generato dal punto *P*, mentre la **Chiave Privata** d è determinata dalla relazione:

d = LOGp (G)

The elliptic curve discrete logarithm problem and ECC Attacks

L'intrattabilità del **Problema del Logaritmo Discreto** è di fondamentale importanza per la sicurezza di qualsiasi algoritmo basato su curva ellittica, ne abbiamo parlato brevemente qualche paragrafo più sopra e concettualmente i problemi connessi al DLP rimangono gli stessi per entrambi i casi. Il DLP applicato alle ECC (**ECDLP**) può essere definito come:

Data una curva E definita su un campo $\mathbb{F}_q$*, un punto* $P \in E(\mathbb{F}_p)$ *di ordine n, ed un punto Q appartenente a <P>, bisogna trovare l'intero l compreso nell'intervallo [1,n-1] tale che Q = l P. Questo intero l è appunto il Logaritmo Discreto di Q in base P, identificabile anche come: = l P. The integer l is called the discrete logarithm of Q to the base P, denoted l = LOGp (Q).*

Capite quindi quanto è importante scegliere dei parametri di dominio adeguati, in modo da poter resistere a tutti gli attacchi basati su ECDLP. Il più diffuso e conosciuto algoritmo per la risoluzione

del ECDLP è l'**Exhaustive Search** (utilizzo la dizione inglese, perchè non si usa tradurre termini del genere), il quale computa la sequenza dei punti P, $2P$, $3P$ fino a trovare Q, il principale svantaggio di questo algoritmo è la lentezza, infatti il Running Time è approssimativamente di n, quindi un n scelto sufficientemente grande ci mette al riparo da questo genere di attacco. Vengono poi gli attacchi più conosciuti e purtroppo o per fortuna maggiormente utilizzati, ovvero il **Pohlig-Hellman** e **Pollard's rho**, che hanno un running time Esponenziale precisamente di $O(Sqrt(p))$ dove p è un numero primo sufficientemente grande, perchè una curva possa resistere a questo genere di attacchi è necessario scegliere una n divisibile per p, cosicchè step $Sqrt(p)$ saranno prettamente inutili dato l'aumento del tempo computazionale.

Il funzionamento del Pollard's Rho è alquanto semplice, dobbiamo eseguire degli step random (meglio definita come **Random Walk**) fino a trovare *ax, ay, bx ,by*, tali che:

$$= ax*B + bx*A == ay*B + by*A$$

$$= ax*l*A + bx*A == ay*l*A + by*A$$

$$= (ax-ay)*l*A == (by-bx)*A$$

$$= (ax-ay)*l == (by-bx) \bmod NP$$

$$= l == (by-bx)*(ax-ay)^{(-1)} \bmod NP$$

(l'esempio e direttamente preso come applicazione dell'attacco che effettuiamo nell'ultimo crackme, quindi se non vi ritrovate con le lettere confrontate quello :))

Troviamo poi una categoria di attacchi denominata **Isomorphism Attacks**, che tentano di ridurre l'ECDLP a DLP, i più conosciuti sono **Weil e Tate Pairing Attacks**, questo genere di attacchi possono essere utilizzati però solo in presenza di **Campi Primi Anomali**.

The Elliptic Curve Digital Signature Algorithm

L'ECDSA è l'algoritmo in curva ellittica corrispondente al DSA, oltre ad essere il più compatto risulta anche essere il più diffuso nelle protezioni di software commerciale e non (i campi sono vasti e variegati in questo caso, come lo sono le piattaforme base che lo supportano), si può anche affermare che è il più "standardizzato" dovendo sottostare a numerose norme, fra le quali **ANSI X9.62, FIPS 186-2, IEEE 1363- 2000** e l' **ISO/IEC 15946-2**. Procediamo adesso elencando passo passo l'algoritmo, attraverso la **Signature Generation** e conseguente **Signature Verification**.

ECDSA signature generation
Input: D = (q,FR, S,a,b, P,n,h)*;* Chiave Privata *d;* messaggio *m.* **Output:**(r,s) 1. Seleziona *k* appartenente ad *r [1,n-1]*. 2. Calcola *kP = (x1, y1)*,

convertendo poi *x1* in un intero *X1*. 3. Calcola *r = X1 mod n*. Se *r = 0*, riseleziona *k* 4. *e = H(m)*. 5. Calcola *s = k^(-1) * (1(e+d*r)) mod n*. Se *s = 0* ricalcola *k*.

ECDSA signature verification
Input: D = (q,FR, S,a,b, P,n,h); Chiave Pubblica *Q*; messaggio *m*, Signature *(r,s)*. **Output:**Accetta / Rifiuta Signature 1. Verifica che *r* ed *s* sono interi compresi nell'intervallo *[1,n-1]*, nel caso la condizione sia falsa restituisce Signature Rifiutata. 2. *e = H(m)*. 3. Calcola *w = s^(-1) mod n*. 4. Calcola *u1 = ew mod n* ed *u2 = rw mod n*. 5. *X = u1P +u2Q*. 6. Se *X = Infinito* restituisce Signature Rifiutata. 7. Converte la coordinata *x1* di *X* in un intero *X1*. 8. Calcola *v = x1 mod n*. 9. Se *v = r* allora la Signature è Valida, altrimenti Rifiuta la Signature.

H() è una qualsiasi funzione di hash, solitamente viene utilizzato *SHA* o *SHA-1*.

A Reverse Engineering Approach

Siamo finalmente arrivati, per la vostra gioia alla parte squisitamente applicativa dell'intero articolo; come target utilizzeremo il crackme 10 di Witeg che implementa l'ECDSA Signature, non mi dedicherò comunque ad analizzare approfonditamente tutti gli aspetti del programma, anche perchè ciò che più mi interessa è mostrarvi direttamente come si ragione i presenza di protezioni basate su ECC.

Il crackme implementa ECDSA non direttamente ma attraverso le librerie MIRACL, c'è da dire che questo è il primo più semplice caso di ECC protection a cui possiamo trovarci, il software professionale non utilizzerà certamente le MIRACL, e non sono per via delle condizioni del copyright, ma semplicemente perchè in poche funzioni (ad esempio .NET o qualunque CryptoApi) i coder possono sbrigarsi. Funzioni esportate da .NET o quant'altro saranno certamente più sicure di un algoritmo costruito passo passo attraverso Miracl, sovente succede però che lo stesso sistema di protezione è male implementato, ad esempio potrete trovare casi in cui alcuni software di sicurezza (importanti fra l'altro!) tendono a ricalcolarsi per ogni utente l'intero Dominio di Parametri..ed altre al fatto che la sicurezza rimane pressocchè invariata (per il fattore Errore Umano tende anzi a scendere) portemmo instaurare delle situazioni di Full Compromise Attack! :))

Supponendo per assurdo che già sappiamo trattarsi di Crittografia su Curva Ellittica, la prima cosa da tenere in considerazione è l'input che l'algoritmo riceve, nel nostro caso ci basta osservare che all'apertura del crackme abbiamo 4 Edit Box, una contenente il Nome e le altre contenenti il Seriale, che si divide in tre campi..probabilmente tre parametri della ECC ;).

Passiamo al secondo fondamentale passo di un qualsiasi ECC, ovvero la **Generazione dei Parametri di Dominio**:

```
miracl *mip;

mip = mirsys(100,10);

mip->IOBASE=16; //La base viene settata a 16, avremo quindi Name e Serial della forma 0..9..A...F

secp160r1_a=mirvar(0); //Coefficiente a

secp160r1_b=mirvar(0); //Coefficiente b

secp160r1_p=mirvar(0); //Punto Base o Generatore

secp160r1_n=mirvar(0); //Ordine del punto base

secp160r1_x=mirvar(0); //Coordinata x

secp160r1_y=mirvar(0); //Coordinata y

cinstr(secp160r1_a, "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFC");

cinstr(secp160r1_b, "1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45");

cinstr(secp160r1_p, "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFF");

cinstr(secp160r1_n, "0100000000000000000001F4C8F927AED3CA752257");
```

```
cinstr(secp160r1_x, "4A96B5688EF573284664698968C38BB913CBFC82");

cinstr(secp160r1_y, "23A628553168947D59DCC912042351377AC5FB32");
```

Come vedete la nostra D sarà: *D(a,b,n,x,y)*, come vedete c'è qualcosa di diverso rispetto al normale, compaiono le coordinate *x* ed *y*, ma non preoccupatevi, più avanti capirete a cosa servono;)
Passiamo adesso all'algoritmo vero e proprio:

```
unsigned long lName,lSNX,lSNR,lSNS,snLSB,i,j;

lName = GetDlgItemTextA(hDlg, EDIT_NAME, szName, 0x40); //lName conterrà il nome

lSNX = GetDlgItemTextA(hDlg, EDIT_X, szSNX, 0x40); //X = primo seriale immesso

lSNR = GetDlgItemTextA(hDlg, EDIT_R, szSNR, 0x40); //R = secondo seriale immesso

lSNS = GetDlgItemTextA(hDlg, EDIT_S, szSNS, 0x40); //S = terzo seriale immesso
```

Ogni curva ellittica necessita di un'inizializzazione:

```
ecurve_init(secp160r1_a,secp160r1_b,secp160r1_p,MR_PROJECTIVE);

epoint* pointG = epoint_init(); //Inizializza un punto della ECC chiamato G

epoint* pointH = epoint_init(); //Inizializza un punto della ECC chiamato H

epoint* pointJ = epoint_init(); //Inizializza un punto della ECC chiamato J

epoint_set(secp160r1_x,secp160r1_y,0,pointG);
```

La prima funzione inizializza una curva ellittica E del tipo y^2 =x^3 + ax + bmod p , ovvero la classica curva che prende anche il nome di Modello di Weierstrass, il quarto parametro (`MR_PROJECTIVE`) specifica se utilizzare le coordinate Affini o Proiettive.

```
hashing(szName, lName, e1); // e1 = H(Name) consultate la Signature Verification

lstrcat(szName, szTag);

lName = lstrlen(szName);

hashing(szName, lName, e2); // e2 = H(Name) questa volta al solo Name è stata
concatenata una Tag statica

cinstr(x, szSNX); // x conterrà il primo serial

cinstr(r, szSNR); // r il secondo serial

cinstr(s, szSNS); // s il terzo serial

snLSB = remain(r,2);
```

Questo semplice pezzo di codice produce i due messaggi e1 ed e2, utilizzando una funzione di hashing H() di cui non ci interessa il funzionamento. La funzione remain() invece divide un big

number con un intero (2 nel nostro caso), trovando così il Resto Intero della divisione r/2, necessaria a fornirci quella serie di bits in più della Point Compression.

```
if ((compare(r,secp160r1_n)

if (point_at_infinity(pointH)==FALSE)
```

Eccoci alla prima Signature Verification, le condizioni controllate sono tre, Se (r < n) e (s<n) e se il punto H infine appartiene alla curva (si lavora con la Point Compression, ecco il perchè del LSB) allora l'esecuzione procede.

Prima Verifica:

```
xgcd(s,secp160r1_n,s,s,s); // s = s^(-1) mod secp160r1_n (s sarebbe la w del punto 3 della SignVer proc ;))

mad(e1,s,s,secp160r1_n,secp160r1_n,u1); // u1= s*e1 mod secp160r1_n

mad(r,s,s,secp160r1_n,secp160r1_n,u2); // u2= s*r mod secp160r1_n

ecurve_mult2(u2,pointH,u1,pointG,pointJ); // J = u1*G + u2*H
```

Questa porzione di codice dovrebbe suonarvi familiare, è infatti la procedura di Signature Verification che abbiamo nei punti 3 - 4 - 5, l'unica difficoltà che potreste incontrare sta nelle lettere diverse ma basta solo un pò di attenzione a non confondersi. Il prossimo controllo è nuovamente sul punto *H*, come avrete capito deve mantenersi diverso da Infinito.

```
epoint_get(pointJ,x,x); // x = J.x in poche parole determina la componente x del punto J

if ((compare(x,z)!=0) && (compare(x,r)==0))
```

L'ascissa del punto J appena trovata, dev'essere uguale ai due messaggi.

Seconda Verifica:

```
mad(e2,s,s,secp160r1_n,secp160r1_n,u1); // u1= s*e2 mod secp160r1_n

mad(r,s,s,secp160r1_n,secp160r1_n,u2); // u2= s*r mod secp160r1_n

ecurve_mult2(u2,pointH,u1,pointG,pointJ); // J = u1*G + u2*H
```

La seconda verifica è praticamente simile alla seconda, ritroviamo ancora una volta il controllo su H ed infine viene determinata ancora una volta l'ascissa del punto J, per poi riavere:

```
if ((compare(x,z)!=0) && (compare(x,r)==0))
```

Se anche questo controllo riesce, la Signature è Corretta!

In conclusione ci troviamo di fronte ad una Signature Verification basata su ECDSA, si tratta quindi di trovare una Signature(*r,s*) e la chiave privata (chiamata in questo caso x) per due messaggi *e1*, *e2* aventi due hash di 160 bits.

Le Verification sono due, una per ogni messaggio ma sono computazionalmente uguali per entrambi i messaggi, possiamo vedere infatti che il nocciolo del nostro controllo è dato dal punto J:

***J** = u1*G + u2*H =*

*= w*e*G + w*r*H =*

*= (w*e + w*r*d)*G*

Dove *w=s^(-1) mod n* mentre *H = d*G*

La J dovrebbe riportarvi alla mente il termine *X* utilizzato nello schema canonico:

***X** = u1P +u2Q*

Ecco che G ed H non sono altro che i famosi *P* e *Q* :)

Come ricorderete il controllo vero e proprio consiste nel verificare che le ascisse dei punti ottenuti devono essere le stesse per i due messaggi, sembra una cosa difficile..ma vi ricordo che siamo nel campo delle ECC, quindo..**Everything is a Point**..!

J(x, y) e la sua proiezione *J(x, -y)* hanno difatti la stessa ascissa, ma impostiamo l'equazione risolutiva:

*w*e1 + w*r*d = -w*e2 - w*r*d*

*= w(e1+e2) = -2*w*r*d*

*d = (n-2)^(-1) * (e1+e2) * r^(-1) mod n*

Che dire le relazioni sono veramente semplici, unica cosa da notare è che ci servirà un piccolo bruteforce per:

*H(r, y1)=k*G*

*J(x, y2)=d*G*

Ormai avrete capito che la coordinata x si può trovare anche come:

x = Lsb(y)

Tenete questo trick bene a mente perchè vi capiterà spessissimo di incontrarlo e/o doverlo usare.

ECDLP and Schoof Solving, for Security Patterns

Passiamo adesso al caso di uno schema che possiede delle debolezze a livello di parametri di dominio, rendendo possibile il Pollard's Rho e/o il Polhig-Hellman per risolvere il DLP.

La risoluzione del Problema del Logaritmo Discreto come già spiegato in precedenza impone di avere almeno il seguenti Parametri:

D(P,Q,a,b,p,np)

Ovviamente ha senso lanciare un ECDLP Attack, se i Parametri sono sufficientemente piccoli e/o sono male implementati, o meglio vengono realizzate delle operazioni o confronti che ci richiamano direttamente a scoprire il DLP. Esistono poi altri casi in cui non è nemmeno necessario utilizzare un DLP ma basta un bruteforce, vi illustro un caso pratico..

```
[...]

004013D0 push eax

004013D1 push 0Ch

004013D3 mov [esp+0A0h+var_58], 0

004013D8 mov esi, ecx

004013DA call bytetobig ;Costruisce un Bignumber A, usando alcune lettere
provenienti da un serial

[...]

004013E7 push ecx

004013E8 push ebx

004013E9 push ebp

004013EA push edi

004013EB call powmod ;27AB8CB1F847BBBC412CAA33^A mod C3CEAB06781ECF3B69EA2103

[...]

0040140C push eax

0040140D push ecx

0040140E call _compare ;Powmod == 7DC79E80D9CBBD7DB291643C
```

Questa porzione di codice è presa da un crackme, è molto semplice ed intuitiva ed allo stesso tempo la più palese indicazione ad utilizzare il DLP :), viene costruito un BigNumber A utilizzando alcuni caratteri del Serial e poi attraverso Powmod computa **27AB8CB1F847BBBC412CAA33^A mod C3CEAB06781ECF3B69EA2103**, e confronta il risultato con **7DC79E80D9CBBD7DB291643C**, di conseguenza per trovare il giusto valore di A dobbiamo usale il DLP!

Ci troviamo nel caso in cui l'ordine di **27AB8CB1F847BBBC412CAA33**, sul campo **F(C3CEAB06781ECF3B69EA2103)** è **C3CEAB06781ECF3B69EA2102 = 2*3*1D*78B*46FA51*89C040BCD81E05** quindi ha senso utilizzare Pollard's Rho e il Polhig-Hellman in combo.

Vediamo ora un altro caso

```
004013F0 push ebp
```

```
004013F1 push esi ;Serial length

004013F2 call inttobig ;Trasforma la lunghezza del serial in BigNum

[...]

004013FD push ebp

004013FE push edi

004013FF call powmod ;27AB8CB1F847BBBC412CAA33^B mod C3CEAB06781ECF3B69EA2103

[...]

00401426 push edx

00401427 push eax

00401428 call _compare ;Powmod == 4A2BEE4544261D982D959675
```

Questo porzione di algoritmo genera un BigNumber *B* contenente la lunghezza del Serial, e come al solito esegue il PowMod **27AB8CB1F847BBBC412CAA33^B mod C3CEAB06781ECF3B69EA2103** confrontando poi il valore con **4A2BEE4544261D982D959675**. Cosa si fa?..Polhig-Hellman..nah perchè scomodarlo *B* pur essendo un BigNumber sarà comunque abbastanza perchè esprime solo la lunghezza del Serial, di conseguenza l'incognita *B* sarà sufficientemente piccola da giustificare un semplice Bruteforce.

Evito adesso di riportarvi il codice e vado direttamente al funzionamento dell'intero algoritmo..

Dopo aver ottenuto *A* e *B*, vengono generati (come non ci interessa) due BigNum *X1* e *X2*, ed in seguito utilizzando la curva $y^2 = x^3 + x$ nel campo **F(ACC00CF0775153B19E037CE879D332BB)** ed attraverso *A* e *B*; vendono determinati:

$P = X1*A + X2*B$

$c = X2 - P.x$

controllando poi che *c* inizi per "TMG-" ed infine copia la parte finale di *c* che reca il Registration Name, *c* sarà quindi nella forma TMG-NAME, considerato ovviamente come BigNum perchè entrerà a far parte della computazione.

Per la risoluzione, si scelgono poi *X1* ed *X2* arbitrari calcolando poi:

$P = X1*A + X2*B$

$X2' = c + P.x$

si rende quindi adesso necessario trovare *X1'* per soddisfare la seguente relazione: **X1*A + X2*B == X1'A + X2'*B** il criterio di equivalenza è posto sul fatto che dobbiamo ottenere la stessa ascissa di *P*. Inoltre *A* e *B* sono dello stesso ordine quindi bisogna calcolare *l* affinchè: $l*A = B$ ovvero bisogna risolvere l'ECDLP. Attraverso Schoof poi possiamo anche sapere che la curva ha **ACC00CF0775153B19E037CE879D332BC** punti, e di conseguenza grazie al **Teorema di Lagrange** si trova l'ordine di *A* e *B* come:

566006783BA8A9D8CF01BE743CE9995E =
2*3*7*D*D*7F*D3*1DF75*5978F*1F374C47*5F73FD8D3

ecco da lontano spuntare il buon caro vecchio Pohlig Hellman, che ci suggerisce di calcolare l'ECDLP come **F(2), F(3), F(7), F(D^2), F(7F), F(D3), F(1DF75), F(1F374C47), F(5F73FD8D3)** .. se permettete e meglio che calcolarlo rispetto a quel numerone intero :)

trovata *l* = **1212121255555ABCDEFABCDEF9999999** segue che:

*X2*A + X1*B == X2'A + X1'*B =*

*= X2*l*B + X1*B == X2'*l*B + X1'*B*

*= X2*l*B + X1*B == X2'*l*B + X1'*B*

*= X2*l + X1 == X2'*l + X1' mod NP*

ed infine..

*X1' = X2*l + X1 - X2'*l mod NP*

Finalmente siamo arrivati alla fine del'articolo, spero di esser stato chiaro e che vi sia piaciuto, come al solito non azzardatevi nemmeno a pensare di mandarmi mail con richieste di crack perchè non vi rispondo proprio (nel migliore dei casi), se invece avete dubbi, difficoltà e quant'altro relativo alla comprensione e/o risoluzione delle ECC scrivetemi pure! :)

Btw Buon 2k7 a tutti!!!!!!!!!!!!!!!!

Note finali

Un caro saluto a: tutta la **UIC**: Quequero, AndreaGeddon, LonelyWolf, PnLuck, ZaiRoN, LittleLuk, **RET**: Devine9, Black-eye, BigS, Aimless, Haldir, ^Daemon^ **ROOTKIT**:BugCheck, Greg Hoglund (Device Driver fuzzing is absolutely killer!;) unchecked IOCTL is the way!), (OpenRCE *) &Pedram (:D), (CounterPane *) &Bruce Shneier and last but not least (my*) (Cattina^*^) for all others not mentioned here..sorry :)

Disclaimer

Noi reversiamo al solo scopo informativo e di miglioramento del linguaggio Assembly.

Capitoooooooo????? Bhè credo di si ;))))